MASTER EN SCIENCES INFORMATIQUES

Prototype de cross-compilateur paramétrable réalisation par une chaine de programmes

Jacquet, J-M

Award date:
1980

Awarding institution:
Universite de Namur

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX
NAMUR

Institut d'informatique

ANNEE ACADEMIQUE I979-I980

P R O T O T Y P E D E
C R O S S - C O M P I L A T E U R
P A R A M E T R A B L E

REALISATION PAR UNE CHAINE DE PROGRAMMES

J.M. JACQUET

Mémoire réalisé en vue
de l'obtention du grade
de Licencié et Maître
en informatique

Je tiens à adresser mes remerciements à toutes les personnes qui ont collaboré, de près ou de loin, à mener à bien ce projet.

Je m'adresse plus particulièrement à:

Monsieur le Professeur Van Bastelaer qui a dirigé le mémoire et assuré la collaboration avec l'U.C.L. dans les meilleures conditions.

Monsieur le Professeur Milgrom qui non seulement m'a permis d'utiliser le matériel de l'U.C.L., mais aussi, grâce à sa disponibilité et ses conseils, m'a permis de mener à bien ce travail.

Monsieur le Professeur Cherton pour les conseils qu'il m'a donné et Monsieur le Professeur Nicoud qui m'a permis d'utiliser le programme UNIASS qu'il a mis au point.

Les membres du personnel de l'unité d'informatique de l'U.C.L. et de l'Institut d'Informatique pour leur accueil et leurs conseils.

Monsieur De Coster qui a participé à la réalisation de ce projet dans le cadre d'un mémoire.

J.M. JACQUET

P L A N

I N T R O D U C T I O N

La mise au point d'un cross-compilateur paramétrable est entreprise au sein du projet X25 étudié en collaboration entre l'U.C.L. à Louvain-la-Neuve et l'Institut d'Informatique de Namur.Ce projet consiste à réaliser un réseau répondant à la norme X25 et utilisant des microprocesseurs comme processeurs frontaux.

Dans le cadre de ce projet,il a semblé intéressant de prévoir dès maintenant la mise au point d'un cross-compilateur paramétrable.Celui-ci doit permettre de traduire des programmes écrits en langage C en codes objets relatifs à différents processeurs.

Dans cette étude,nous avons essayé de construire un premier prototype de cross-compilateur paramétrable en utilisant une chaîne de programmes pour le réaliser.Il a été mis au point dans l'espoir qu'il puisse être facilement utilisable pour différents processeurs.Cependant,il a été essentiellement testé pour le microprocesseur Motorola 6800.Certaines phases du cross-compilateur ont également été testées pour le microprocesseur Intel 8080.

Dans ce texte,nous nous proposons,dans un premier chapitre,de définir les buts essentiels d'un cross-compilateur ainsi que ses principes de réalisation sur base d'une chaîne de programmes.Ensuite,nous présentons les outils utilisés - langages et logiciels - pour le réaliser.Le chapitre suivant décrit de façon précise les différentes étapes de réalisation du cross-compilateur pour le microprocesseur Motorola 6800.Enfin,dans un dernier chapitre,nous analysons les éléments du processeur qui influencent le cross-compilateur et donnons les conseils nécessaires à sa mise en oeuvre pour un autre processeur.

C H A P I T R E I

BUTS ESSENTIELS ET PRINCIPES DE REALISATION

D'UN CROSS-COMPILATEUR PARAMETRABLE

Dans ce chapitre,nous définissons tout d'abord les principaux termes qui sont utilisés dans cet exposé:

compilateur
cross-compilateur
cross-compilateur paramétrable
pré-compilateur
langage assembleur
programme assembleur
cross-assembleur
cross-assembleur paramétrable

Nous présentons ensuite les buts et avantages essentiels d'un cross-compilateur paramétrable en soulignant également ses limites.

Enfin,nous décrivons les solutions de réalisation envisageables par une chaîne de programmes ainsi que la solution retenue.Dans ce troisième point nous expliquons aussi,brièvement,pourquoi nous avons retenu une solution comportant une chaîne de programmes.

I.I Définition des termes utilisés

Nous allons définir les principaux termes utilisés dans la suite de cet exposé.

I.II compilateur

programme permettant de traduire un programme écrit dans un langage de haut niveau (différent d'un langage de type assembleur) en code objet exécutable sur un processeur.

I.I2 cross-compilateur

programme permettant de traduire un programme écrit dans un langage de haut niveau en code objet exécutable sur un processeur différent du processeur qui exécute ce cross-compilateur.

I.I3 cross-compilateur paramétrable

cross-compilateur permettant de générer les codes objets relatifs à différents processeurs.

I.I4 pré-compilateur

programme permettant de traduire un programme écrit dans un langage de haut niveau en un autre langage symbolique plus élémentaire et plus simple à manipuler.

I.I5 langage assembleur

langage permettant à l'utilisateur d'un ordinateur de ne pas écrire ses programmes dans le langage primitif de la machine.Il lui offre la possibilité de désigner sym-

boliquement les entités fondamentales qu'il doit manipuler: instructions, constantes, variables, emplacements mémoires, textes d'instructions.

I.I6 programme assembleur

programme permettant de traduire un programme écrit dans un langage de type assembleur en code objet exécutable sur un processeur.

I.I7 cross-assembleur

programme permettant de traduire un programme écrit dans un langage de type assembleur en code objet exécutable sur un processeur différent du processeur qui exécute ce cross-assembleur.

I.I8 cross-assembleur paramétrable

cross-assembleur permettant de générer les codes objets relatifs à différents processeurs.

I.2 Buts d'un cross-compilateur paramétrable

Dans ce paragraphe,nous présentons les buts essentiels et les avantages que peut offrir un cross-compilateur paramétrable.Ces avantages sont particulièrement importants dans le cadre d'un réseau de microprocesseurs de types différents. Ils peuvent également s'avérer très intéressants pour un processeur unique.

Un des principaux avantages d'un cross-compilateur paramétrable est qu'il ne nécessite que la connaissance d'un seul langage de programmation de haut niveau -en l'occurrence le langage C.

Ceci facilite évidemment la tâche des programmeurs puisqu'ils ne devront pas écrire leurs programmes dans le langage assembleur du processeur où ces programmes devront être exécutés - ou éventuellement dans d'autres langages utilisables sur ce processeur.

Il faut cependant noter,dès maintenant,que dans le but d'obtenir des progammes exécutables plus performants,les programmes C devront tenir compte,dans une certaine mesure, des particularités propres au processeur.

D'autre part,la paramétrisation du cross-compilateur ne sera pas élémentaire à réaliser et demandera un assez long travail de mise au point.

Un autre avantage qu'on peut souligner est que la probabilité de portabilité des programmes est plus grande.Ceci s'avère vrai aussi bien dans le cadre d'un réseau que dans celui d'une extension du système informatique ou lors d'un changement de matériel.

Dans le cadre d'un réseau - où les différents noeuds réalisent les mêmes fonctions - les programmes utilisés dans les différents noeuds seront fort semblables même s'ils sont destinés à des processeurs de types différents.Il suffira de les adapter en tenant compte de certaines particularités du processeur.

On peut également retenir que la portabilité des programmes permettra de simplifier la maintenance et la mise au point des programmes.Ceux-ci peuvent,étant donné qu'ils sont tous écrits dans le même langage,être compris par tous les programmeurs.Cela assurera donc une maintenance plus rapide,plus efficace et plus sûre.

Lors d'un changement de matériel,par exemple,il suffira de reprendre les programmes C existants et de les adapter au nouveau processeur.

I.3 Solutions de réalisation envisageables par une chaîne de programmes

Dans ce paragraphe, nous expliquons brièvement pourquoi nous avons choisi une solution utilisant une chaîne de programmes. Nous décrivons ensuite les différentes solutions envisageables et celle qui a été retenue. Nous donnons également les raisons essentielles de ce choix.

I.3I Pourquoi une chaîne de programmes

Le cross-compilateur paramétrable que nous construisons est un premier prototype. Son but essentiel est de distinguer les problèmes qui peuvent surgir lors de l'élaboration d'un cross-compilateur paramétrable.

L'utilisation d'une chaîne de programmes pour le réaliser permet de sérier les problèmes en les classant en différents types. Elle permet également une approche plus modulaire du problème. Ceci permet éventuellement de réétudier certaines parties du cross-compilateur qui seraient jugées mal adaptées ou peu performantes.

I.32 Solutions de réalisation envisageables

Avant de décrire plus en détail le cross-compilateur tel qu'il a été réalisé, voyons brièvement quelles étaient les solutions de réalisation possibles pour construire un tel cross-compilateur par une chaîne de programmes. Deux des solutions envisagées comportent trois phases essentielles, la troisième n'en comprend que deux.

I.32I Solutions en trois phases

Chacune des deux solutions en trois phases utilise une première phase de pré-compilation.Celle-ci consiste en une traduction de chaque instruction C en une suite d'instructions d'un langage symbolique plus élémentaire que nous appelerons langage intermédiaire.

La deuxième phase du traitement consiste à traduire chaque instruction de ce langage intermédiaire dans un langage de type assembleur.

Dans la première solution,ce code assembleur est un code universel et standardisé,c'est-à-dire que sa syntaxe est identique quel que soit le processeur concerné.Dans ce cas,la troisième phase consiste en un cross-assemblage paramétrable qui en fonction du processeur sur lequel le programme doit être exécuté,traduit le code assembleur généré lors de la phase précédente en code objet exécutable sur ce processeur (voir figure I.I).

Dans la seconde solution,la deuxième phase traduit les instructions du langage intermédiaire directement dans le code assembleur relatif au processeur sur lequel le programme sera exécuté.La troisième phase consiste ici en un cross-assemblage utilisant directement le programme d'assemblage standard du processeur concerné. (voir figure I.2)

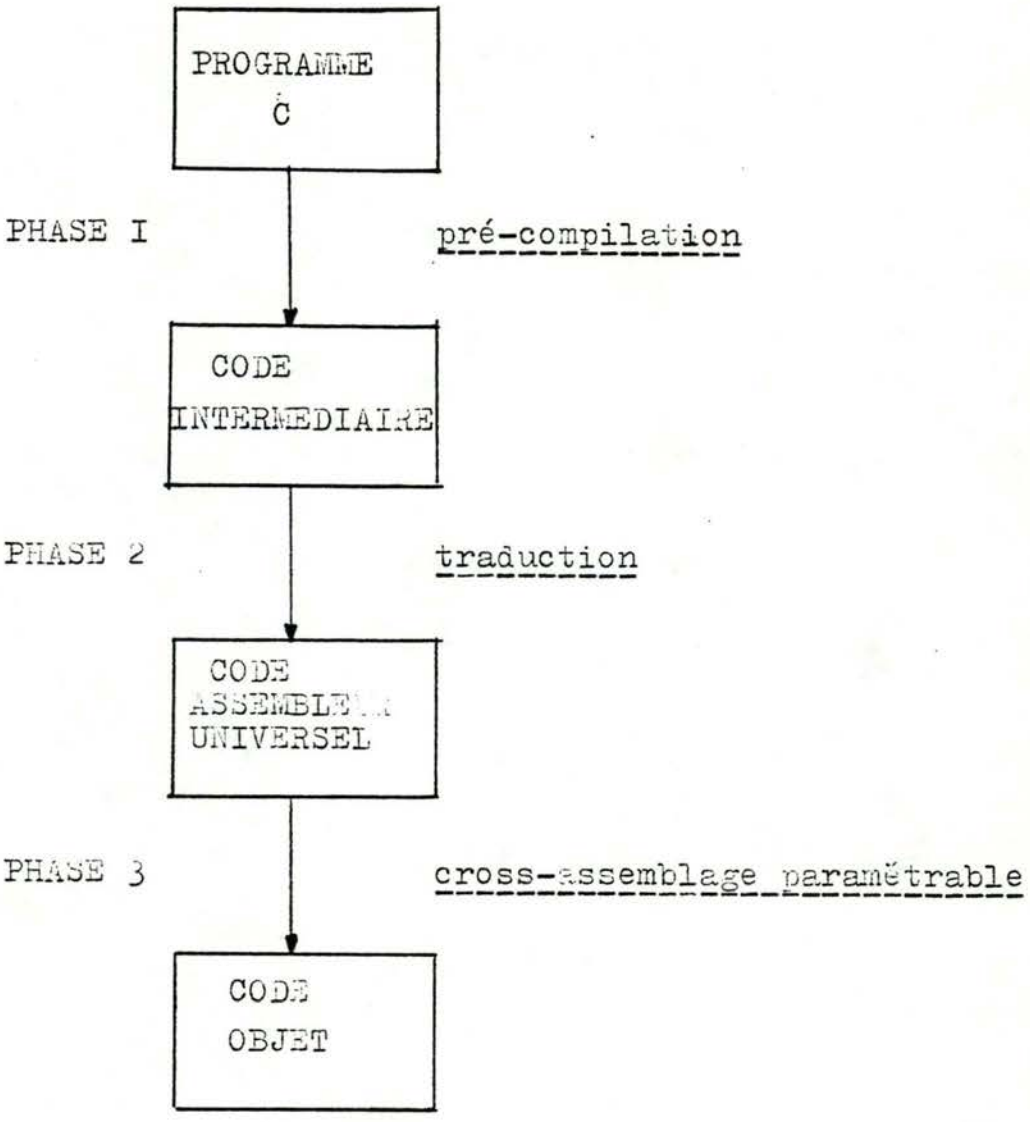


figure I.I première solution d'uncross-compilateur paramétrable en trois phases

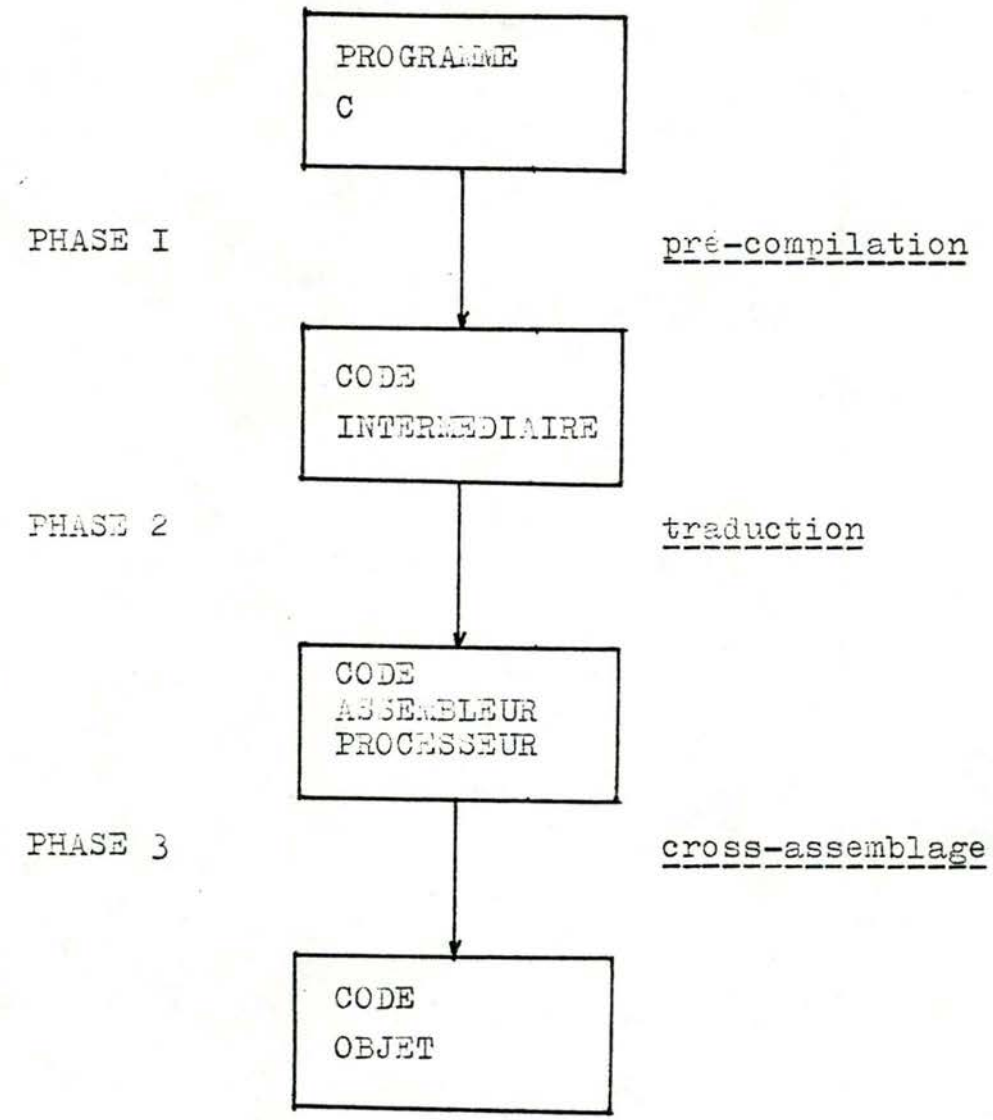


figure I.2 deuxième solution d'un cross-compilateur paramétrable en trois phases

I.322 Solution en deux phases

Cette troisième solution utilise la même phase de pré-compilation que les deux autres solutions. Rappelons que cell-ci consiste à traduire chaque instruction C en une suite d'instructions d'un langage symbolique plus élémentaire.

La deuxième phase est ici une cross-compilation paramétrable traduisant directement le code obtenu lors de la phase de pré-compilation en code objet relatif au processeur qui doit exécuter le programme traduit. (voir figure I.3)

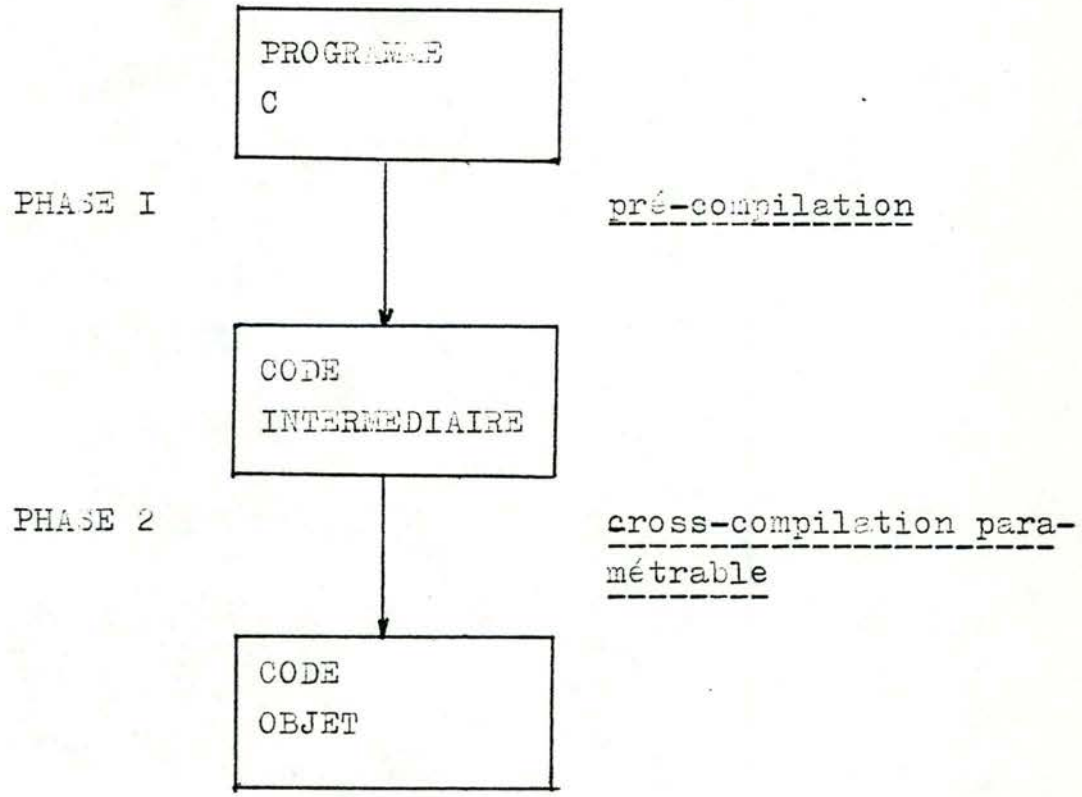


figure I.3 solution du cross-compilateur paramétrable en 2 phases

I.33 Solution retenue et raisons de ce choix

La solution que nous avons décidé d'implémenter comprend 3 phases .Il s'agit de la première solution décrite dans le paragraphe précédent.

Elle comprend:

- une phase de pré-compilation
- une phase de traduction en assembleur universel
- une phase de cross-assemblage paramétrable

(voir figure I.I)

Voyons maintenant quelles sont les raisons essentielles de ce choix.

Parmi les solutions que nous venons d'envisager,la solution comportant deux phases(pré-compilation et cross-compilation paramétrable) semble intéressante.En effet,elle comporte une phase de traitement de moins que les deux autres ce qui permet de diminuer le temps total de traitement.D' autre part,cette phase de cross-compilation paramétrable aurait sans doute pu être réalisée sur base du cross-assembleur paramétrable dont nous disposions.

Mais connaissant mal cet outil,nous avons préféré,par manque de temps,ne pas le modifier.En plus,cette solution est plus compliquée à réaliser en ce sens qu'elle mélange deux problèmes différents:d'une part une phase de traduction dans un langage assembleur,d'autre part la génération du code objet correspondant à ce langage assembleur.

Nous avons donc rejeté cette solution.

La raison essentielle qui nous a poussé à nous tourner vers la solution utilisant un cross-assembleur paramétrable est qu'il suffit d'écrire et de connaître un seul programme pour réaliser cette phase de cross-assemblage.

En effet,la solution utilisant le langage assembleur standard du processeur nécessite d'écrire et d'apprendre à manipuler un nouveau programme de cross-assemblage pour chaque processeur.Par contre,dans la solution retenue,il suffit de parametrer le programme de cross-assemblage existant.

D'autre part,l'utilisation d'un langage assembleur universel plutôt que le langage assembleur standard du processeur offre des avantages certains.Elle permet de faciliter la paramétrisation du cross-compilateur.En effet, la définition des séquences d'instructions à générer pour chaque instruction du langage intermédiaire généré par la phase de pré-compilation sera simplifiée,en ce sens que les instructions assembleurs répondent toutes à la même syntaxe quel que soit le code objet du processeur.

En plus,nous désirions tester un cross-assembleur paramétrable que nous avions reçu de l'Ecole Polytechnique Fédérale de Lausanne.

C H A P I T R E II

DESCRIPTION DES OUTILS

UTILISES

Dans ce chapitre,nous allons essentiellement décrire les outils que nous avons utilisés pour réaliser le cross-compilateur paramétrable.

Nous décrivons tout d'abord les langages dont nous nous sommes servis:

- Le langage C
- Le code VAC
- Le code CALM
- Snobol 3

Ensuite,nous décrivons le logiciel existant qui a été utilisé:

- Le compilateur VAC
- Le cross-assembleur paramétrable UNIASS

Nous pouvons ensuite compléter le schéma du cross-compilateur paramétrable que nous avons retenu (voir figure I.I) et donner un exemple concret traduisant une instruction C en code objet pour un processeur.

II.I Langages utilisés

C est un langage de programmation initialement développé sur le système d'exploitation UNIX tournant sur DEC PDP-II.Il a été réalisé et implémenté par Dennis Ritchie. Des compilateurs C sont également disponibles sur un certain nombre d'autres machines,notamment l' IBM/370,Honeywell 6000 et Interdata 8/32.

La plupart des idées reprises pour développer le langage C proviennent du langage BCPL développé par Martin Richards.Cette influence de BCPL sur C a également conduit indirectement au développement du langage B qui a été écrit par Ken Thompson en I970 pour être utilisé sur le premier système UNIX tournant sur PDP-7.

C possède la plupart des avantages des langages de haut niveau.Voyons quelles en sont les caractéristiques essentielles.

C dispose d'un grand nombre de types de données.Il permet l'utilisation de caractères,d'entiers,de nombres en virgule flottante,de pointeurs,de tableaux,de structures et de fonctions.

C reprend la plupart des instructions de contrôle que l'on retrouve dans les langages de haut niveau.Ces instructions permettent une structuration bien adaptée des programmes.Ces instructions sont:if,while,for,do,switch.

Toutes les routines de C peuvent être définies récursivement grâce à l'existance d'une pile où sont sauvées automatiquement toutes les informations nécessaires à la restitution de l'ancien environnement.

C ne dispose pas de la structure de blocs.Un programme C comprend une ou plusieurs procédures.Cela permet la modularité des programmes;mais les procédures ne peuvent pas être emboîtées.Cela permet d'éviter la mise en place de mécanismes complexes d'accès.On peut également noter que chaque procédure peut être compilée séparément.

C ne dispose pas de tableaux dynamiques.Ceci permet une organisation simple de la pile,la place nécessaire pour stocker les données étant connue au moment de la compilation. Mais l'existance de pointeurs permet de définir des procédures qui peuvent travailler sur des tableaux de longueurs différentes:le pointeur donne accès au premier élément du tableau.

Les arguments d'une procédure sont appelés par valeur. C'est-à-dire qu'ils sont passés à la fonction appelée en copiant la valeur du paramètre.Il est ainsi impossible à la procédure appelée de modifier la valeur du paramètre actuel de la procédure appelante.Pour réaliser un appèl par référence,il est cependant possible d'utiliser un pointeur,ce qui permet à la procédure appelée de modifier l' objet sur lequel pointe ce pointeur.Les noms de tableau sont passés comme la localisation du premier élément du tableau.Il s'agit donc effectivement d'un appèl par référence.

Ces quelques considérations sur le langage C montrent que,bien qu'offrant la plupart des avantages des langages de haut niveau,il possède la particularité de conserver des techniques d'implémentation relativement simples.

D'autre part,le langage C repond aux possibilités des machines actuelles.La plupart des types de données disponibles et des structures de contrôle sont supportés par

la plupart des ordinateurs.C est donc relativement efficace et l'utilisation d'un langage assembleur pour écrire les programmes s'avère moins évidente.La meilleure preuve est qu'une grande partie du système UNIX est écrite en C.

Bien que n'étant pas un langage de très haut niveau, C reste cependant indépendant de l'architecture de la machine.Il est ainsi facilement possible d'écrire des programmes C qui soient portables sur des machines différentes.

Pour toutes ces raisons,avantages des langages de haut niveau,possibilité de manipulation des mêmes objets -caractères,nombres,adresses- que les langages assembleurs,portabilité,nous avons décidé d'utiliser C comme langage de base et d'écrire un cross-compilateur paramétrable de C.

Il est également évident que,travaillant sur le système UNIX,nous désirions utiliser tous les programmes écrits en C dans le système d'exploitation.

II.I2 Le code VAC (Varian Abstract C-Machine)

Le code VAC a été réalisé lors du projet TRIX étudié en collaboration entre l'U.C.L. de Louvain-la-Neuve et la K.U.L. de Leuven.Ce projet visait à l'étude et l'implémentation de systèmes d'exploitations modulaires pour des mini-ordinateurs.Ces systèmes d'exploitations doivent être,dans la mesure du possible,indépendants du hardware.Des versions opérationnelles devaient fonctionner sur les mini-ordinateurs des deux institutions - un PDP II/45 à Louvain-la Neuve et un Varian 73 à Leuven - Pour réaliser cet objectif,il était nécessaire d'utiliser un langage indépendant de la machine. Le choix s'est porté sur le langage C.

Afin de pouvoir utiliser C sur la Varian 73,une machine-C abstraite a été conçue.Elle est orientée vers l'utilisation d'une pile.

Nous allons maintenant décrire cette machine-C.Dans une première partie nous décrivons comment est organisée la mémoire.La deuxième partie décrit l'environnement lors de l'exécution.La troisième partie montre brièvement les instructions VAC disponibles.

II.I2I La mémoire

Cette section nous aidera à comprendre l'organisation des objets manipulés par les instructions VAC ainsi que la gestion des variables disponibles.

La mémoire a été organisée de la façon suivante:elle contient un segment de code et un segment de données (voir figure II.I).

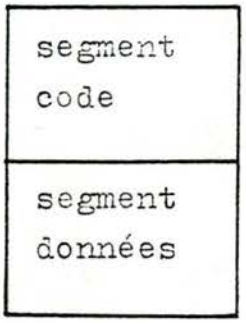


figure II.I organisation de la mémoire centrale

Le segment de code est le code virtuel généré.
Le segment de données est divisé en deux parties. (voir figure II.2)

Dans la partie permanente sont stockés tous les objets qui existent pendant l'entièreté du programme.Ces objets sont adressés relativement à une adresse de base globale (pointeur g).La longueur de cette zone permanente est connue à la compilation.

Dans la partie stack sont stockés les objets qui existent uniquement pendant l'exécution d'une procédure.Une nouvelle zone est créée à l'entrée de chaque procédure et détruite à la sortie.

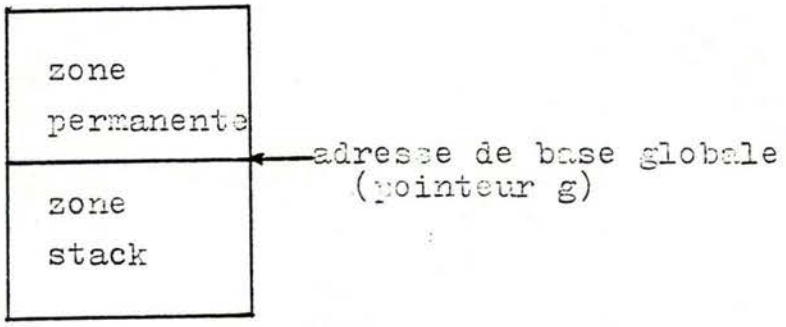


figure II.2 organisation du segment de données

Voyons maintenant comment est organisée cette pile. Elle comprend les paramètres,une zone link,les variables locales et des objets temporaires.(voir figure II.3)

Les variables locales,les paramètres et la zone link sont adressés relativement à une adresse de base locale (pointeur l) qui est calculée à l'entrée de chaque procédure.

Les objets temporaires sont adressés par rapport à un stack pointer (pointeur s).

La zone link est définie comme étant l'adresse de base locale (pointeur l) et le stack pointer (pointeur s) de la

procédure appelante ainsi que l'adresse de retour à cette procédure.

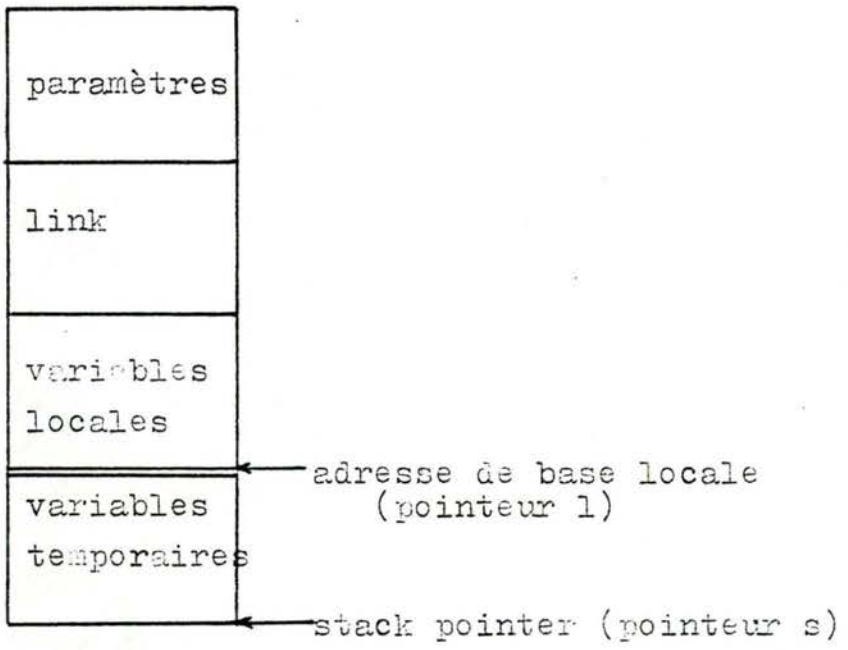


figure II.3 organisation de la pile

II.I22 Environnement d'exécution

Dans le paragraphe précédent,nous avons donné une image statique de l'environnement d'exécution.Nous décrivons ici les changements d'environnement.

Cette partie nous permettra de comprendre les opérations à réaliser lors d'une modification d'environnement.

Le nouvel environnement à créer lors d'un appel de procédure est réalisé en deux étapes:une séquence d'appel dans la procédure appelante et une séquence d'entrée dans la procédure appelée.L'ancien environnement est restitué dans la séquence de retour de la procédure appelée.Décrivons

brièvement ces trois séquences.

II.I22I Séquence d'appel

Trois opérations sont réalisées dans la séquence d'appel:

- réserver une place au sommet de la pile pour y placer le résultat éventuel de la procédure appelée.
- placer les paramètres au sommet de la pile.
- appeler la procédure.

II.I222 Séquence d'entrée

Deux opérations essentielles sont réalisées dans la séquence d'entrée:

- sauver l'ancien environnement dans la zone link (adresse de base locale, stack pointer et adresse de retour de la procédure appelante).
- calculer la nouvelle adresse de base locale de la procédure appelée. (Il faut noter qu'à l'entée d'une procédure, le stack pointer est identique à l'adresse de base locale) .

II.I223 Séquence de retour

La séquence de retour réalise les fonctions suivantes:

-placer le résultat éventuel de la procédure appelée à la place réservée à cet effet lors de la séquence d'appel.
- restaurer l'adresse de base locale et le stack pointer de la procédure appelante et brancher à l'adresse de

retour de cette même procédure.

II.I23 Jeu d'instructions VAC

On peut trouver en annexe des tableaux reprenants l'ensemble de ces instructions.(voir annexe I) Nous nous bornons ici à en citer les grandes classes.

- instructions de modification de l'environnement
- instructions de manipulations de valeurs
- instructions de manipulations d'adresses
- opérations binaires
- opérations unaires
- opérations d'incrément et de décrément
- instructions de branchements
- instructions switch

II.I3 Le code CALM (Common Assembly Language for Microprocessors)

Le code CALM a été mis au point par le laboratoire de calculatrices digitales de l'Ecole Polytechnique de Lausanne.

Il s'agit d'un langage de type assembleur qui a été construit en comparant les langages assembleurs de treize microprocesseurs:Intel 8080,Zilog 80,Motorola 6800,Mos 650x, Signatics 2650,RCA I802,FAIR F8,NS-SC/MP,EA 9002,II I000, NS PACE,II 9900,DEC-LSI II.Il possède un jeu d'instructions standard et est utilisable pour un grand nombre de processeurs.

Son rôle essentiel est de permettre l'écriture de tout programme selon une syntaxe identique quel que soit le processeur utilisé.Il est cependant évident que des caracté-

ristiques d'architecture telles que les registres manipulables par CALM varient de processeur à processeur.

Un autre avantage est la clarté de sa syntaxe avec notamment l'absence du mode d'adressage dans les mnémonics des instructions.Il n'apparaît,en effet,que dans le champs des opérandes.

Nous verrons les conventions et les règles de CALM dans le chapitre III.On remarquera que ces conventions sont fort semblables à celles des langages assembleurs conventionnels.Il apparaîtra également immédiatement que le jeu d'instructions proposé par CALM est plus lisible et plus compréhensible que celui de la plupart des assembleurs des microprocesseurs.

Pour illustrer ceci,donnons à titre d'exemple,la syntaxe de quelques instructions des microprocesseurs Motorola 6800 et Intel 8080 et comparons la à celle proposée par CALM.

Les deux tableaux qui suivent illustrent respectivement la syntaxe standard et CALM pour les instructions d'addition de l'Intel 8080 et la syntaxe standard et CALM pour les instructions d'addition du Motorola 6800.

r = A,B,C,D,E,H,L (registres 8 bits)
(HL) (adresse dans registre Iб bits HL)
M (adresse mémoire)
data = valeur immédiate 8 bits
rp = BC,DE,HL,SP (registres Iб bits)

Intel 8080	CALM	opération
ADD r	ADD A,r	(A)+(r) ->A
ADC r	ADDC A,r	(A)+(carry)+(r) ->A
ADI data	ADD A, data	(A)+data ->A
ACI data	ADDC A, data	(A)+(carry)+data ->A
DAD rp	ADD HL,rp	(HL)+(rp) ->HL

r = #data (valeur immédiate 8bits)
M (adresse mémoire)
disp,X ou (IX)+disp (adressage indexé avec déplacement disp relatif au registre d'index respectivement pour le langage standard et le langage CALM
A et B = accumulateurs

Motorola 6800	CALM	opération
ADDA r	ADD A,r	(A)+(r) ->A
ADDB r	ADD B,r	(B)+(r) ->B
ADCA r	ADDC A,r	(A)+(carry)+(r) ->A
ADCB r	ADDC B,r	(B)+(carry)+(r) ->B
ABA	ADD A,B	(A)+(B) ->A

A titre d'exemple,nous donnons également en annexe le jeu d'instructions standard et son équivalent CALM pour le Motorola 6800 (voir annexe 2.A et 2.B)

II.I4 Snobol 3

Snobol 3 a été mis au point par les Bell Telephone Laboratories. Il s'agit d'un langage de programmation qui est une généralisation et une extension du langage Snobol qui a été développé dès I962.

Snobol 3 permet une utilisation et une manipulation aisée des chaînes de caractères.

Il permet de référencer symboliquement des chaînes de caractères et dispose de mécanismes permettant facilement de rechercher, de former ou de réarranger des chaînes de caractères.

L'aspect essentiel qui nous intéresse dans Snobol 3 est la possibilité de générer facilement du code symbolique en fonction de la reconnaissance d'une chaîne de caractères.

Donnons en dès maintenant un exemple concret:

```
(I)  suit0      input = syspit
(2)             input   'PLUS'      /f(suitI)
(3)             syspot = 'POP A'
(4)             syspot = 'POP B'
(5)             syspot = 'ADD A,B'
(6)  suitI
```

Expliquons cette suite d'instructions. Notons tout d'abord que syspit est la fonction de lecture du fichier input standard tandis que syspot est la fonction d'écriture sur le fichier output standard.

L'instruction (I) permet donc de lire une ligne du fichier input standard et de mettre le résultat dans la zone input.

L'instruction (2) regarde si la zone input commence par les caractères 'PLUS' .Si ce n'est pas le cas,elle branche à l'étiquette 'suitI' .

Les instructions (3),(4)et(5) permettent d'écrire 3 lignes dans le fichier output standard.Après l'exécution de ces trois instructions celui-ci comprend:

```
POP A
POP B
ADD A,B
```

Après l'instruction (5),l'instruction (6) est exécutée.

II.2 Logiciels utilisés

II.2I Le pré-compilateur VAC

Le pré-compilateur VAC a été réalisé,comme le code VAC lors du projet TWIX dont nous avons déjà parlé.Rappelons que ce projet,dans le but de mettre en oeuvre des systèmes d'exploitation pour mini-ordinateurs - PDP II/45 et Varian 73 - a conçu une C-machine lui permettant notamment d'utiliser C sur la Varian 73.

Le pré-compilateur VAC permet de transformer un programme écrit en C en un nouveau programme écrit en code VAC (correspondant a la C-machine).

Ce pré-compilateur a été construit à partir du compilateur C original.Nous nous proposons,ici,d'en expliquer brièvement la structure générale (voir figure II.4).

Le pré-compilateur VAC est composé de 3 processus:

- un processus d'enchaînement des opérations et de contrôle.
- un processus d'analyse du programme C.
- un processus de traduction en code VAC.

II.2II Processus d'enchaînement des opérations et de contrôle

Ce processus réalise les opérations suivantes:

- lecture et vérification des paramètres
- préparation des noms des fichiers temporaires et des fichiers résultats
- activation du pré-processeur C

Ce pré-processeur C est un macro-générateur permettant d'inclure un fichier dans le programme source et de remplacer dans le texte d'un programme toutes les occurrences de certains symboles par des valeurs de substitution.

- activation du processus d'analyse du programme C
- activation du processus de traduction en code VAC

Ce processus est appelé deux fois:une première fois pour générer le code et un segment de donnees (non constantes),une deuxième fois pour générer les constantes.

- concaténation des différents fichiers résultats pour créer le fichier résultat final.

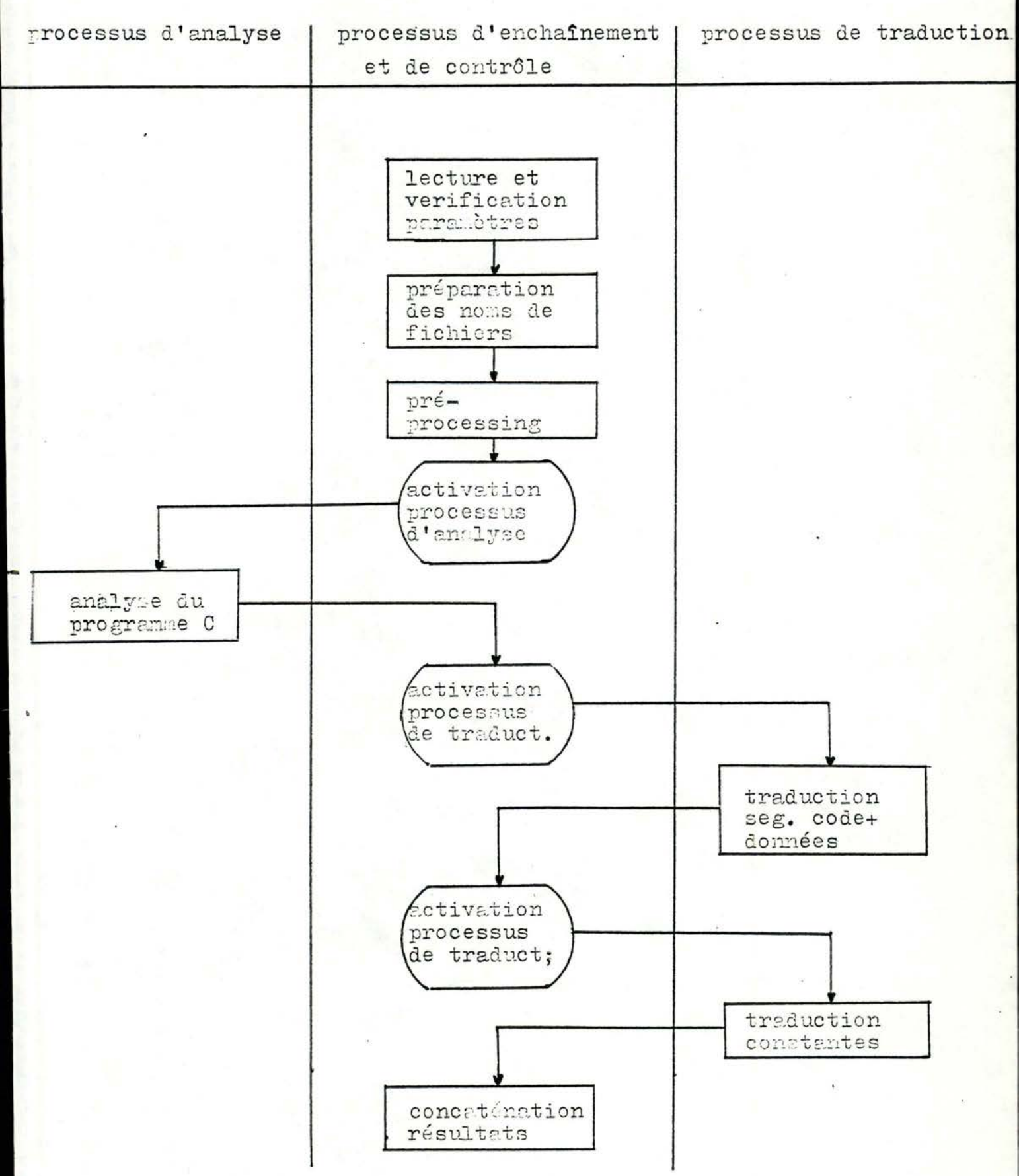
II.2I2 Processus d'analyse du programme C

Ce processus effectue la compilation proprement dite en vérifiant la syntaxe du programme C.Il traduit le programme C dans un langage intermédiaire interne au compilateur et crée la table des symboles.

II.2I3 Processus de traduction en code VAC

Ce processus de traduction reprend les résultats fournis par la phase d'analyse du programme C et les traduit en code VAC.comme nous l'avons déjà dit,ce processus est appelé deux fois.La première fois pour générer le segment de code et un segment de données,la seconde fois pour générer les constantes.

figure II.4 structure générale du pré-compilateur VAC



II.22 Le cross-assembleur paramétrable UNIASS

UNIASS est un cross-assembleur paramétrable qui a été écrit en Pascal par M. Schmit au Laboratoire de minis et micros-ordinateurs de l'Ecole Polytechnique Fédérale de Lausanne. Il fonctionne sur CDC CYBER, PDP IO, PDP II.

Il permet de générer du code binaire pour tout langage assembleur dont on lui fournit une description. Il s'agit d'un programme capable de reconnaître et de sauver une description correspondant à un processeur donné et de générer le code objet pour les instructions du langage assembleur qui correspondent à la description fournie.

UNIASS, tel qu'il a été conçu, est essentiellement destiné à générer du code binaire et non du code symbolique (même si cela est réalisable). De plus, la version existante ne permet pas de générer plus de 9 bytes par instruction. Il semble cependant que ceci est facilement extensible.

La description à fournir à UNIASS n'est pas une description de l'architecture de la machine mais seulement une description formelle du langage assembleur de la machine utilisée. Les outils de description offerts par UNIASS seront décrits dans un chapitre ultérieur. Nous nous bornons ici à expliquer la structure générale du fonctionnement de UNIASS.

Le cross-assembleur UNIASS est logiquement divisé en deux phases qui peuvent opérer indépendamment ou simultanément:

- un générateur de structure d'arbre qui sauve les informations contenues dans la description du processeur.
- un assembleur qui analyse chaque ligne du programme

source de l'utilisateur en suivant le chemin correspondant dans la structure d'arbre et qui génère le code objet qui y correspond si la ligne est syntaxiquement correcte.

La structure d'arbre pourrait être générée une fois pour toute et sauvée pour des utilisations ultérieures.On parle alors du mode compilation qui permet d'accélérer le cross-assemblage (voir figure II.5) - il faut noter que ce mode compilation n'est pas implémenté.

La description peut également précéder le programme source.On parle alors du mode interpréteur.Il permet à l'utilisateur d'écrire lui-même ses propres descriptions en fonction de ses besoins spécifiques (voir figure II.6).

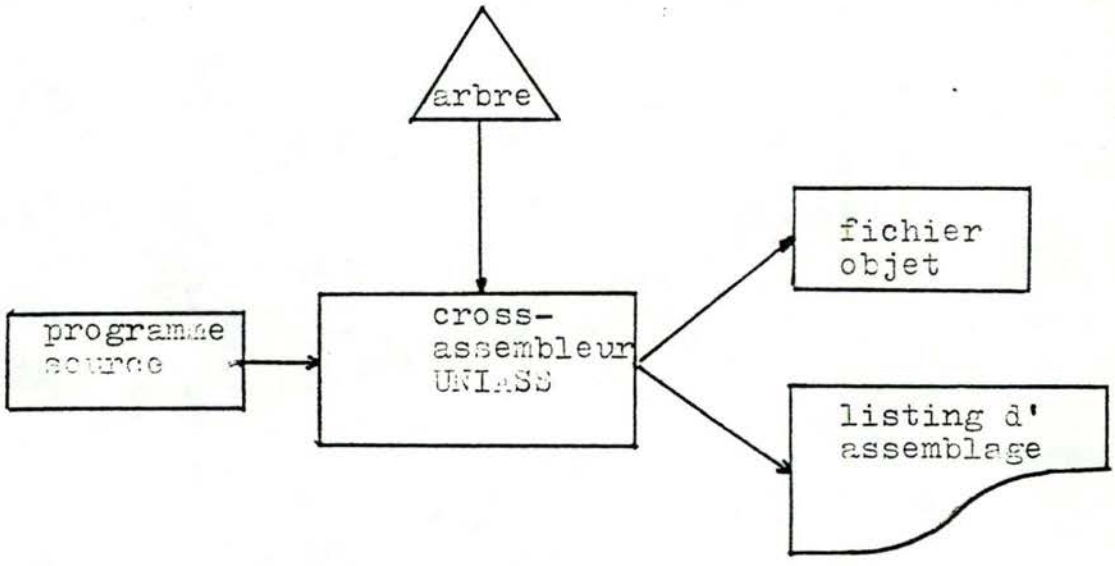


figure II.5 mode compilation

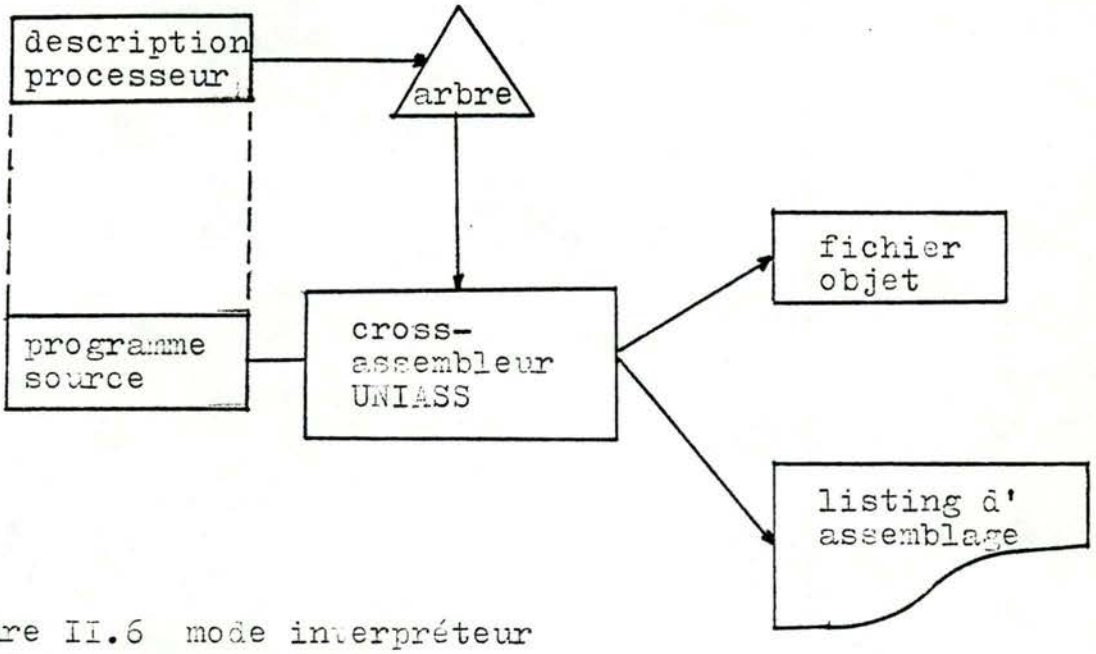
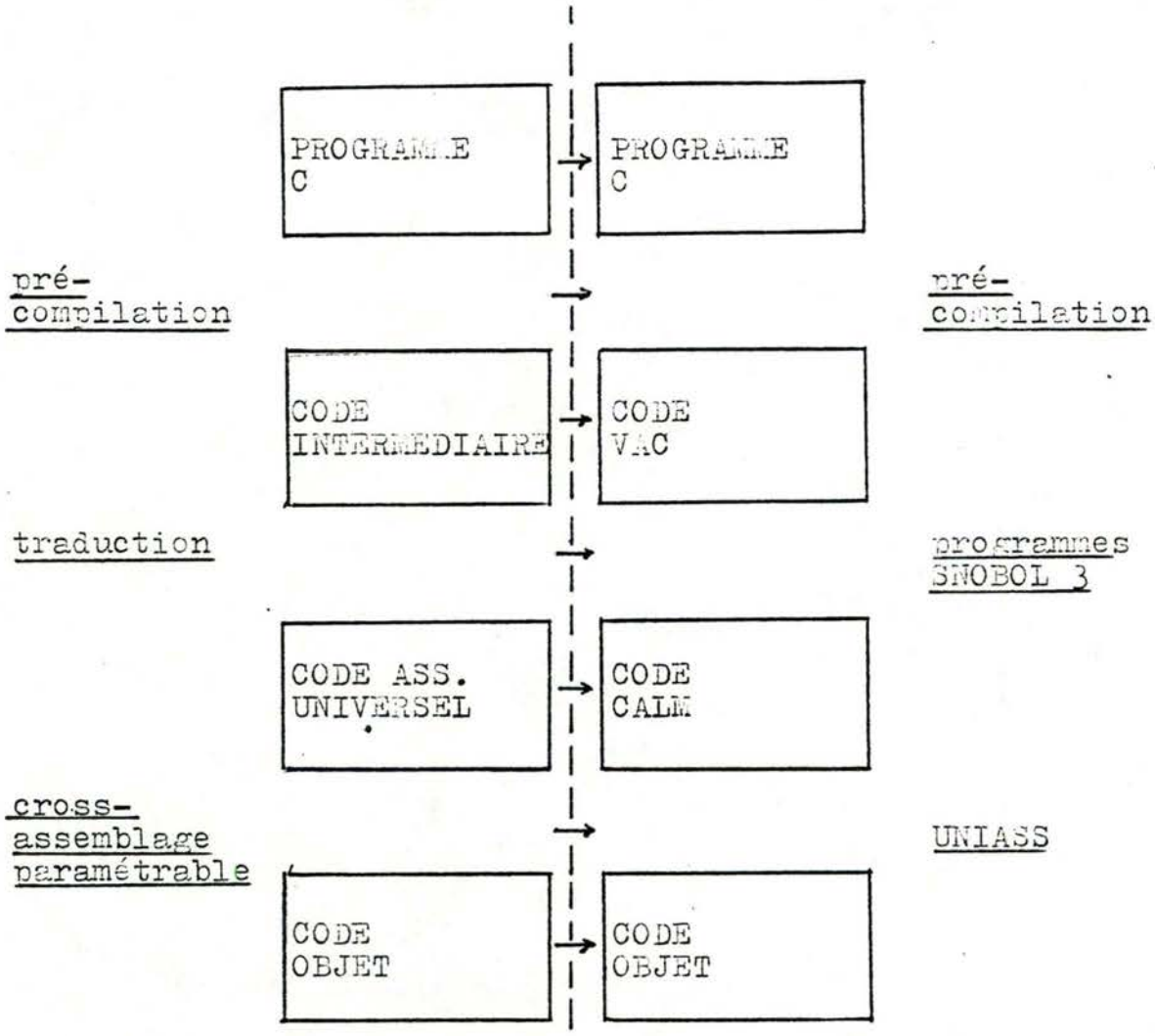


figure II.6 mode interpréteur

II.3 Complément du schéma du cross-compilateur paramétrable et exemple concret.

Après avoir décrit les langages et les logiciels que nous allons utiliser pour réaliser le cross-compilateur, nous sommes à même de préciser le schéma de ce cross-compilateur (schéma de la figure I.I).

Reprenons et complétons ce schéma.



Nous voyons, au niveau des langages que le code intermédiaire correspond au code VAC tandis que le code assembleur universel est le code CALM.

Donnons à présent un exemple concret de traduction d'une instruction C en code objet pour le Motorola 6800.

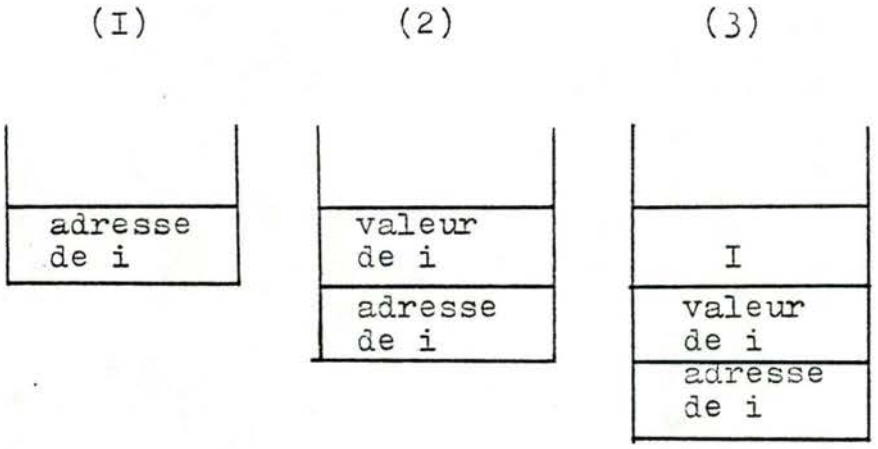
Soit l'instruction C : i=i+I (i est une variable entière)

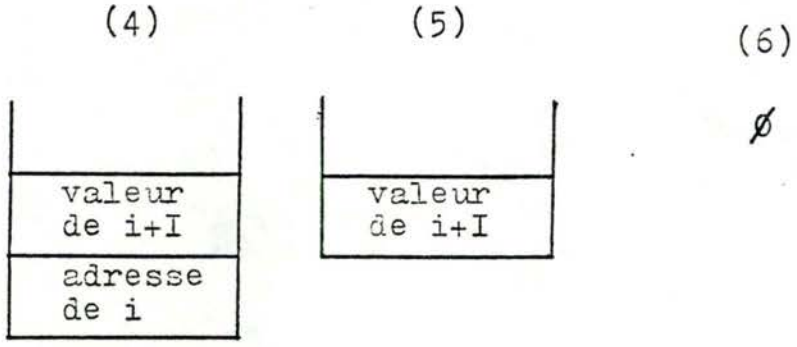
- traduction de C vers VAC

Le code VAC obtenu est le suivant:

```
(I)  LALOC 0     ;place l'adresse de i au sommet de la pile
(2)  LVLOCI 0    ;place la valeur de i au sommet de la pile
(3)  LVCONI I    ;place la constante I au sommet de la pile
(4)  PLUSI       ;additionne les 2 éléments au sommet de la
                  pile et y sauve le résultat
(5)  SVALI       ;affecte la valeur au sommet de la pile
                  à l'adresse également sur la pile
(6)  REMVEI      ;enlève la valeur au sommet de la pile
```

Montrons l'évolution de la pile pour bien comprendre cet exemple.





- traduction de VAC vers CALM

Chacune des 6 instructions VAC ci-dessus est ensuite traduite en code CALM.

Le code CALM obtenu est le suivant:

LALOC 0 devient:

```
(I)  LOAD A,LL
(2)  LOAD B,LH          ;place l'adresse de base locale
                         dans A et B
(3)  LOAD IX,#2*0       ;place le déplacement par rapport à
                         l'adresse de base locale dans IX
(4)  LOAD SH,IX         ;SH et SL (adresse SH+I) contiennent
                         le déplacement
(5)  ADD A,SL
(6)  ADDC B,SH          ;calcul de l'adresse de i
(7)  PUSH A             ;sauve A sur la pile
(8)  PUSH B             ;sauve B sur la pile
```

LVLOCI 0 devient:

```
(9)  LOAD IX,LH         ;place l'adresse de base locale
                         dans IX
(I0) LOAD B,(IX)+2*0
```

```
(II) LOAD A,(IX)+I+2 0;place la valeur de i dans A et B
(I2) PUSH A          ;sauve A sur la pile
(I3) PUSH B          ;sauve B sur la pile
```

LWCONI I devient:

```
(I4) LOAD IX, I      ;place I dans IX
(I5) LOAD SH,IX      ;SH et SL contiennent (IX)
(I6) LOAD A,SL
(I7) LOAD B,SH       ;A et B contiennent (IX)
(I8) PUSH A          ;sauve A sur la pile
(I9) PUSH B          ;sauve B sur la pile
```

PLUSI devient:

```
(20) POP B
(2I) POP A           ;extrais l'opérande 2 dans A et B
(22) INC IX,SP       ;IX pointe sur l'opérande I
(23) ADD A,(IX)+I
(24) ADDC B,(IX)+0   ;additionne les 2 opérandes
(25) PUSH A          ;sauve A sur la pile
(26) PUSH B          ;sauve B sur la pile
```

SVALI devient

```
(27) POP B
(28) POP A           ;extrais la valeur au sommet de la
                      pile
(29) INC IX,SP       ;IX pointe sur l'adresse de i
(30) INC SP
(3I) INC SP          ;incrémente le stack pointer
(32) LOAD IX,(IX)+0  ;place l'adresse de i dans IX
(33) LOAD (IX)+I,A
```

```
(34) LOAD (IX)+0,B      ;place la valeur extraite à l'
                         adresse de i
(35) PUSH A             ;sauve A sur la pile
(36) PUSH B             ;sauve B sur la pile
```

REMVEI devient:

```
(37) POP B
(38) POP A              ;enlève l'élément au sommet de la
                         pile
```

On peut déjà noter dès à présent que certaines opérations se présentant à la fin d'une instruction V.6 et au début de la suivante sont inutiles.Il s'agit des séquences:

```
PUSH A
PUSH B
POP B
POP A
```

Nous verrons dans le chapitre III que ces séquences sont supprimées

- traduction de CALM vers le code objet Motorola 6800

Nous obtenons en supposant que LH se trouve à l'adresse 0 et LL à l'adresse I, que SH se trouve à l'adresse 2 et SL à l'adresse 3:

(I) 226 00I
(2) 326 000
(3) 3I6 000 000
(4) 337 002
(5) 233 003
(6) 33I 002
(7) 066
(8) 067
(9) 336 000
(I0) 346 000
(II) 246 00I
(I2) 066
(I3) 067
(I4) 3I6 000 00I
(I5) 337 002
(I6) 226 003
(I7) 326 002
(I8) 066
(I9) 067

(20) 063
(2I) 062
(22) 060
(23) 253 00I
(24) 35I 000
(25) 066
(26) 067
(27) 063
(28) 062
(29) 060
(30) 06I
(3I) 06I
(32) 356 000
(33) 247 00I
(34) 347 000
(35) 066
(36) 067
(37) 063
(38) 062

N.B.: Le code objet est représenté en octal.

Le premier byte correspond au code opératoire (3 digits).

Le ou les bytes suivants sont les adresses des opérandes ou des valeurs immédiates.

C H A P I T R E III

MISE EN OEUVRE DU CROSS-COMPILATEUR

POUR LE MOTOROLA 6800

Dans ce chapitre,nous décrivons le cross-compilateur paramétrable tel qu'il a été réalisé pour le Motorola 6800.

Nous présentons dans un premier point la phase de pré-compilation,en rappelant son but et l'outil de base utilisé.Nous décrivons ensuite les modifications apportées au pré-compilateur existant.Enfin,nous donnons un exemple de traduction de C vers VAC.

Le second point décrit la phase de traduction de VAC vers CALM.Nous rappelons le but de cette phase et l'outil utilisé(SNOBOL 3).Nous expliquons pourquoi nous avons utilisé SNOBOL 3.Nous décrivons ensuite les deux passes de cette phase de traduction,une passe de traduction et une d'optimisation du code obtenu.Enfin,nous donnons un exemple de traduction de VAC vers CALM.

Le troisième point décrit la phase de cross-assemblage.Nous rappelons le but et l'outil utilisé pour réaliser cette phase.Nous donnons ensuite les règles de description pour paramétrer UNIASS avec des exemples.Nous donnons également un exemple de traduction en code objet pour le Motorola 6800.

III.I Phase de pré-compilation

Rappelons que la phase de pré-compilation a pour but de transformer un programme écrit en C en un autre programme en code VAC.L'outil utilisé pour réaliser cette pré-compilation est le pré-compilateur VAC qui a été décrit en II.2I.

Il faut noter que nous avons apporté un certain nombre de modifications au pré-compilateur existant.

Dans un premier temps,nous allons expliquer les raisons de ces modifications.Nous décrivons ensuite ces modifications avant de fournir un exemple de traduction de quelques instructions C en code VAC.Nous donnons également un exemple complet de traduction d'un programme C.

III.II Raisons des modifications apportées

Comme nous l'avons déjà précisé,le pré-compilateur VAC a été réalisé en fonction de la Varian 73.Certaines de ses particularités étaient mal adaptées ou peu performantes pour notre problème.Il nous a donc semblé intéressant d'apporter un certain nombre de modifications au pré-compilateur existant.

Une partie de ces modifications a permis d'utiliser directement la syntaxe de CALM plutôt que celle de la Varian 73.Ceci est particulièrement vrai pour certaines directives,(réservation de bytes,de mots,...) pour les étiquettes et les commentaires et permet de ne plus les modifier lors de la phase de traduction en code CALM.

Les autres modifications permettent d'améliorer les performances,en obtenant des programmes V.C plus concis, en définissant de nouvelles instructions mieux adaptées, en introduisant le type de données caractère.Elles permettent un gain appréciable de place mémoire et une amélioration des temps d'exécution des programmes cross-compilés.

Nous allons maintenant décrire chaque modification.

III.I2 Modifications apportées au pré-compilateur V.C

III.I2I Modification des commentaires

Certains commentaires,concernant directement la Varian 73 ont été supprimés.Les autres ont été modifiés afin de répondre à la syntaxe de CALM.Tout commentaire commençait par un "*".Ce caractère a été remplacé par le caractère ";".

III.I22 Modification des directives de réservation d'emplacements mémoires

Ces modifications ont également été apportées pour répondre aux propres directives de CALM.

Il s'agit des directives:

- DATA valeur remplacée par
  - .BYTE valeur
  - .WORD valeur

Cette directive-DATA-réservait un mot de I6 bits et lui affectait la valeur "valeur".

Les deux nouvelles directives-.BYTE et .WORD-réservent respectivement un byte et un mot de I6 bits et lui affec-

tent la valeur "valeur".

Le fait que nous générons deux directives s'explique car le pré-compilateur existant ne connaissait que le type entier.Nous verrons plus loin que le type caractère a été ajouté.Il fallait donc une directive capable de réserver des bytes.

- BSS valeur remplacée par
  .BLKB valeur

Cette directive réserve "valeur" bytes sans initialisation.

III.I23 Modification des étiquettes

Les étiquettes avaient la syntaxe suivante dans l'ancien pré-compilateur:

$Li EQU

Cette étiquette est devenue:

Li:

III.I24 Introduction du type caractère

Comme nous l'avons signalé en III.I22,le seul type de données connu en VAC est le type entier.Cela implique que tout caractère est transformé en entier et occupe 16 bits en memoire d'où une perte importante de place.

Cela est dû au fait que la Varian 73 ne connait que le type entier.

Il nous est apparu important d'introduire le type caractère au sein du pré-compilateur afin d'obtenir un gain substantiel de place mémoire.

III.I25 Introduction d'instructions de manipulation des caractères

A partir du moment où nous introduisons le type caractère,il est nécessaire d'introduire les instructions VAC qui permettent de manipuler ces caractères.

Les instructions introduites sont au nombre de cinq:

LVLOCC
LVEXTC
INDIRC
SVALC
LVPARC

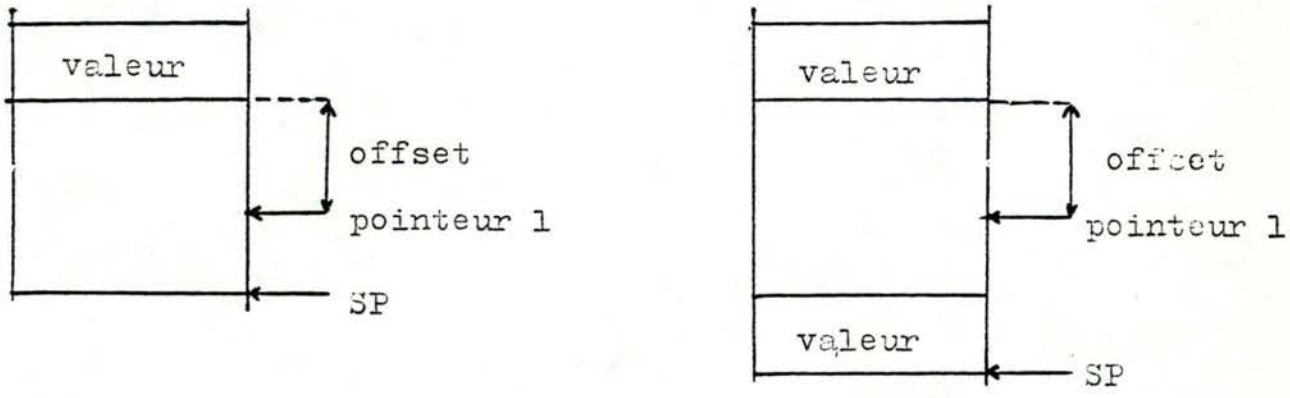
Décrivons ces instructions.

- LVLOCC (Load Value LOCal for Characters)

format: LVLOCC offset

Cette instruction permet de placer un objet de type caractère au sommet de la pile.Le paramètre offset est le déplacement de l'objet en question par rapport à une adresse de base locale (pointeur 1) dont nous avons parlé et expliqué le rôle en II.I2I (voir figure II.5)

offset de l'instruction:



AVANT APRES

-LVPARC (Load Value of PARameter for Characters)

format: LVPARC offset

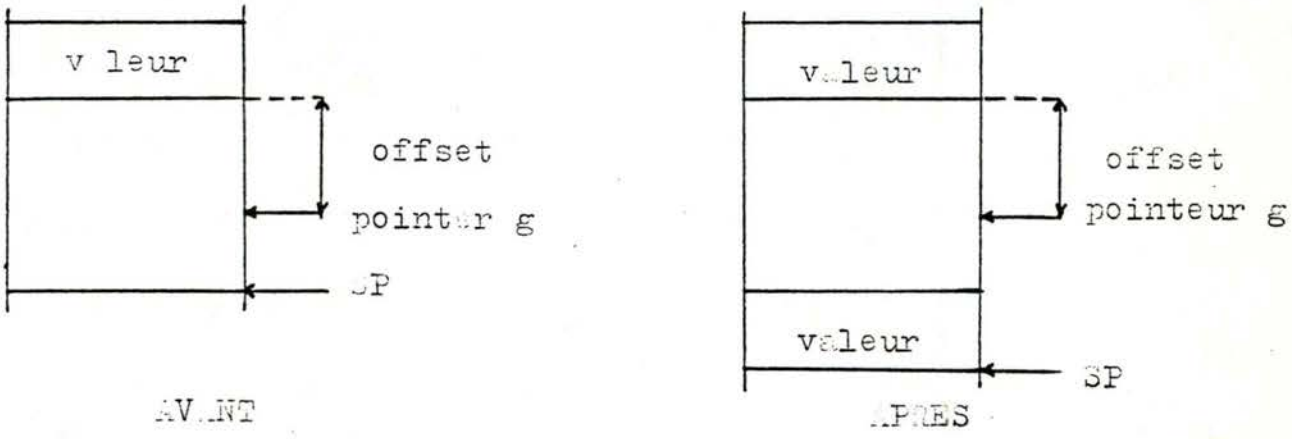
Cette instruction a le même effet que LVLOCC mais est utilisée pour placer les paramètres (de type caractère) d'une fonction au sommet de la pile.

- LVEXTC (Load Value EXTernal for Characters)

format: LVEXTC offset

Cette instruction a le même effet que LVLOCC mais le paramètre offset fournit le déplacement par rapport à l'adresse de base globale (pointeur g) dont nous avons expliqué le rôle en II.I2I (voir figure II.2)

effet de l'instruction:

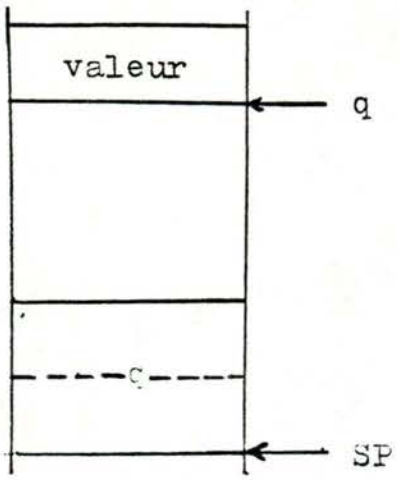


- INDIRC

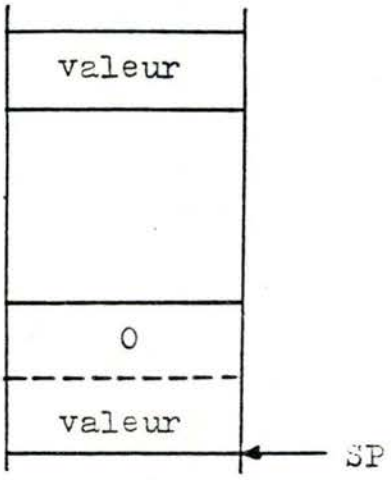
format: INDIRC

L'entier se trouvant au sommet de la pile est interprété comme une adresse et est remplacé par la valeur (caractère) sur laquelle il pointe.

effet de l'instruction:



AVANT



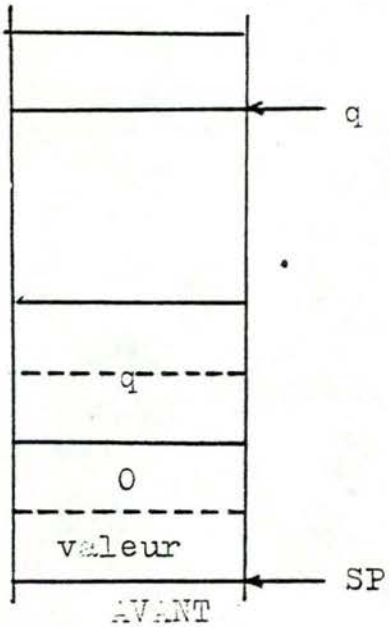
APRES

- SVALC

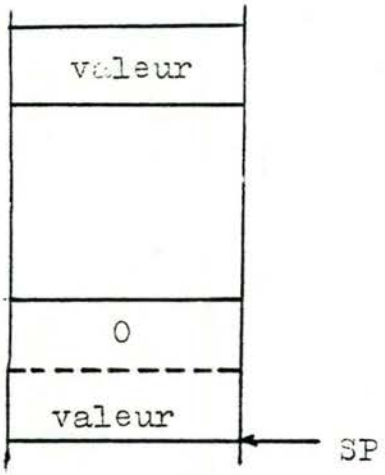
format: SVALC

Cette instruction permet de modifier la valeur d'une variable par assignation. L'adresse et la nouvelle valeur (caractère) sont toutes deux au sommet de la pile. Après exécution, la valeur de la variable est changée. Seule l'adresse est enlevée du sommet de la pile.

effet de l'instruction:



AVANT



APRES

III.I26 Suppression des branchements inutiles

Le pré-compilateur,tel qu'il était conçu ne permettait pas de détecter certains branchements inutiles qui avaient la forme suivante:

```
     JUMP étiquette
étiquette
     instruction suivante
```

C'est-à-dire que l'adresse de branchement suit directement l'instruction de branchement.Cette instruction JUMP est évidemment inutile.Par contre l'étiquette doit être conservée car elle peut être utilisée par une autre instruction.

La nouvelle version du pré-compilateur ne génère plus de telles instructions de branchement.

III.I27 Regroupement d'instructions

Il est apparu que certaines séquences d'instructions VAC pouvaient être regroupées pour former de nouvelles instructions plus sophistiquées mais permettant de générer moins d'instructions CALM lors de la phase de traduction en code CALM,d'où un gain ultérieur de place mémoire et une amélioration du temps d'exécution des programmes cross-compilés.

Ces séquences d'instructions sont:

```
GEI
JUMPF label
```

```
LTI
JUMPT label
```

```
GTI
JUMPF label
```

```
LEI
JUMPF label
```

LEI JUMPF label	GTI JUMPT label
LTI JUMPF label	GEI JUMPT label
EQI JUMPF label	NEI JUMPT label
NEI JUMPF label	EQI JUMPT label

Dans toutes ces séquences d'instructions, certaines opérations sont effectuées deux fois. En effet, la première instruction (GEI, GTI, LEI, LTI, EQI, NEI) effectue un test entre les deux valeurs au sommet de la pile (respectivement $\geq, >, \leq, <, ==, !=$) et met le résultat (1 si le résultat du test est vrai, 0 sinon) au sommet de la pile. JUMPF et JUMPT, quand à eux, effectue un branchement à l'étiquette label conditionnellement à la valeur de l'élément se trouvant au sommet de la pile. JUMPF branche à label si la valeur de cet élément est nul, JUMPT si sa valeur est différente de 0.

Le même test est donc réalisé deux fois. Nous créons donc 6 nouvelles instructions qui réalisent ces opérations.

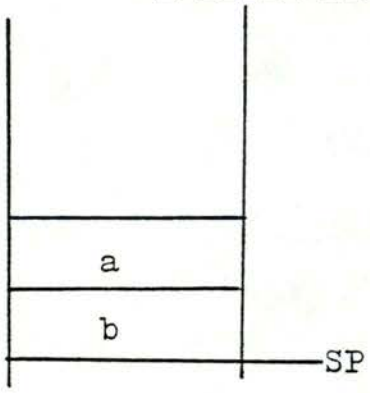
Il faut noter qu'il faut 6 nouvelles instructions et non pas 12 car chaque séquence d'instructions VAC se trouvant en parallèle ci-dessus est identique.

Spécifications de ces instructions

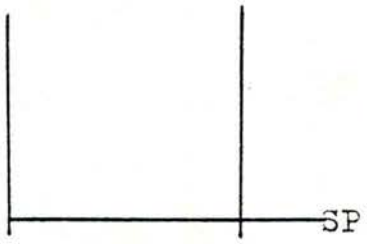
Ces instructions sont:

[JLE, JGE, JLT, JGT, JNE, JEQ] label

effet sur la pile



AVANT



APRES

instruction	rôle	séquences correspondantes	
JLE label	a≤b branche à label sinon, en séquence	LEI JUMPT label	GTI JUMPF label
JGE label	a≥b branche à label sinon, en séquence	GEI JUMPT label	LTI JUMPF label
JLT label	a<b branche à label sinon, en séquence	LTI JUMPT label	GEI JUMPF label
JGT label	a>b branche à label sinon, en séquence	GTI JUMPT label	LEI JUMPF label
JNE label	a!=b branche à label sinon, en séquence	NEI JUMPT label	EQI JUMPF label
JEQ label	a==b branche à label sinon, en séquence	EQI JUMPT label	NEI JUMPF label

III.I3 Exemples de pré-compilation

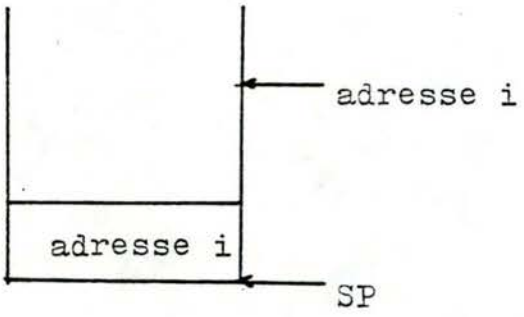
Nous allons maintenant donner quelques exemples de traduction d'instructions C en code VAO.Nous donnons également un exemple reprenant un programme C complet.

III.I3I Exemples pour quelques instructions C

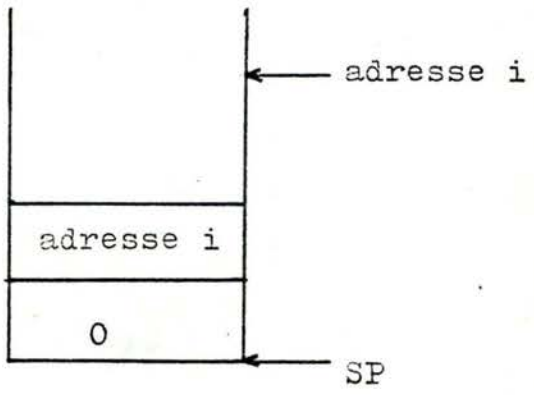
- instruction d'affectation

i=0 (i entier) devient:

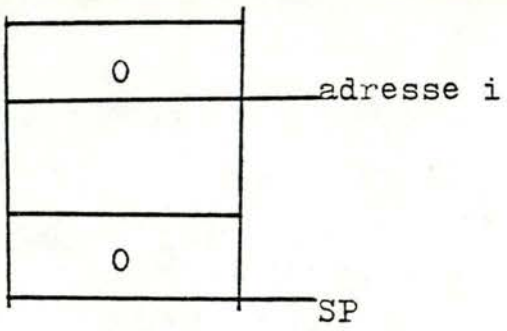
```
(I) LALOC 0   ;place l'adresse de i au sommet de
                la pile
(2) LVCONI 0  ;place la constante 0 au sommet de
                la pile
(3) SVALI     ;affecte la valeur au sommet de la
                pile à l'adresse également sur la
                pile
(4) REMVEI    ;retire la valeur au sommet de la
                pile
```



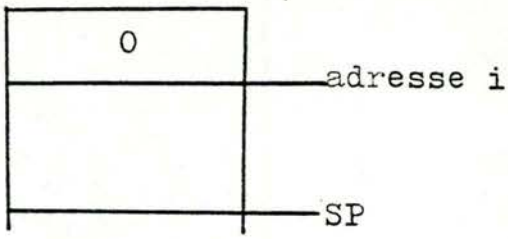
(I)



(2)



(3)

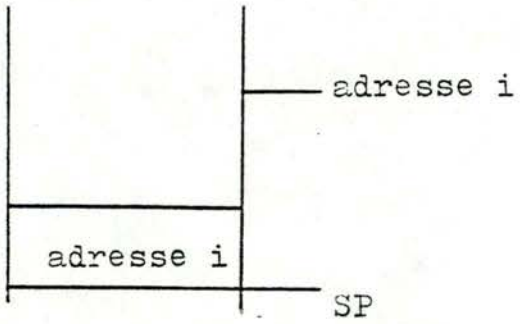


(4)

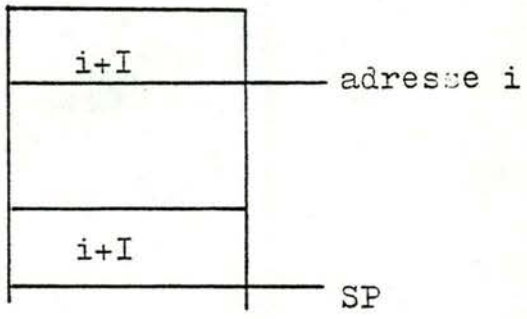
- instruction de post-incrément

i++ (i entier) devient:

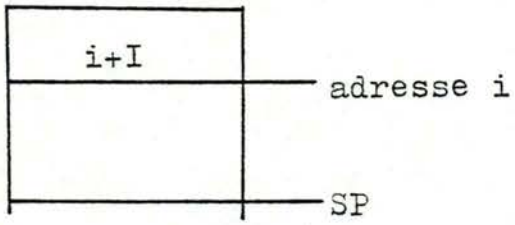
(I) LALOC 0 ;place l'adresse de i au sommet de la pile
(2) INCAFT I ;incrément de i
(3) REMVEI ;retire l'élément au sommet de la pile



(I)



(2)

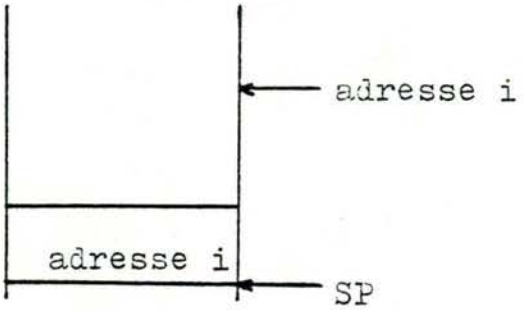


(3)

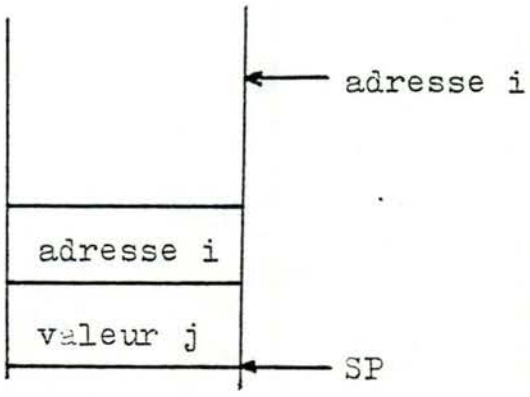
- instruction d'addition

i=j+I (i,j entiers) devient:

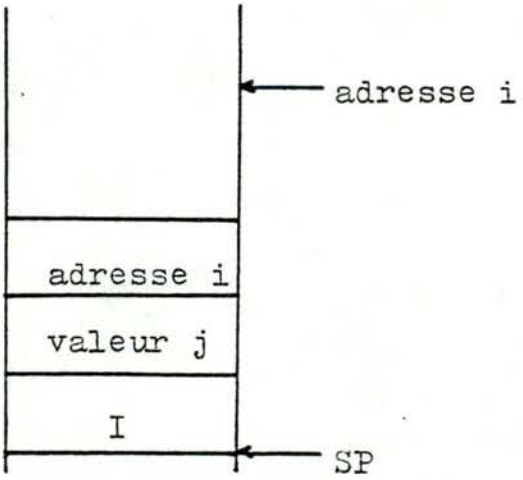
```
(I) LALOC I    ;place l'adresse de i au sommet de
                la pile
(2) LVLOCI 0   ;place la valeur de j au sommet de
                la pile
(3) LVCONI I   ;place la constante I au sommet de
                la pile
(4) PLUSI      ;addition
(5) SVALI      ;affecte le résultat à i
(6) REMVEI     ;retire l'élément au sommet de la
                pile
```



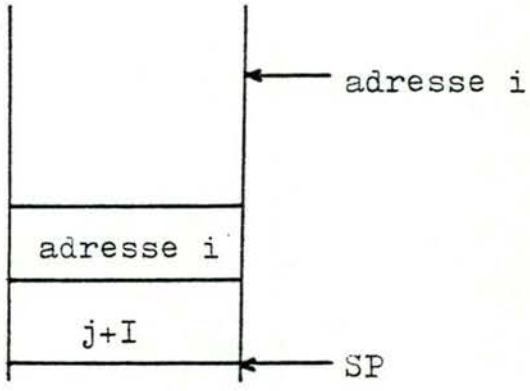
(I)



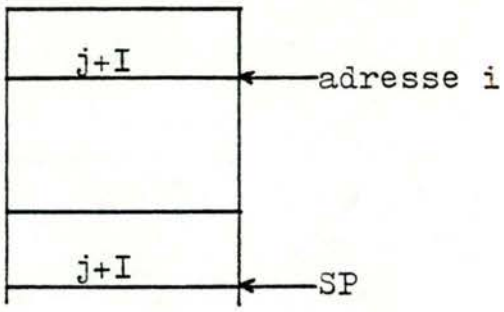
(2)



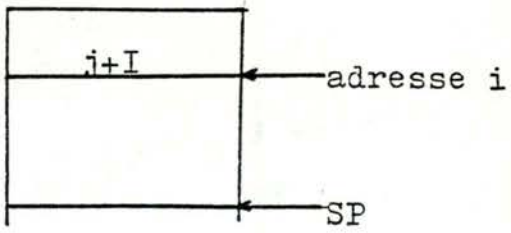
(3)



(4)



(5)



(6)

III.I.2 Exemple de programme C

Ce programme, élémentaire, effectue la somme des nombres de I à 50.

Il faut noter que les commentaires en regard de chaque instruction V.C ne sont pas générés automatiquement. Ils ont été ajouté pour expliquer l'exemple.

```
#
/*      Ce programme additionne les nombres
        de 0 a MAX (50)                                 */


#define ZERO    0
#define MAX     50
main() {
        int i,tot;
        tot=0;
        for(i=ZERO; i<=MAX; i++)
                tot=tot+i;
}
```

```
text
IN      NAME    MAIN
        ENTRY   ;adresse de base locale au sommet de la pile
        i=-1 + NLOC
        tot=-2 + NLOC
        SAVE    2       ;calcul de l'adresse de base locale de la fonction MAIN


        new expression at line 11       ;expression tot=0

        LALOC   0       ;adresse de tot au sommet de la pile
        LVCONI  0       ;0 au sommet de la pile
        SVALI           ;affectation tot=0
        REMVEI          ;enlever l'element au sommet de la pile
        new expression at line 12       ;expression i=ZERO

        LALOC   1       ;adresse de i au sommet de la pile
        LVCONI  0       ;0 au sommet de la pile
        SVALI           ;affectation i=0
        REMVEI          ;enlever l'element au sommet de la pile


        new expression at line 12       ;expression i<=MAX

        LVLOCI  1       ;valeur de i au sommet de la pile
        LVCONI  50      ;50 au sommet de la pile
        JGT     L3      ;branchement a L3 si i>50

        new expression at line 13       ;expression tot=tot+i

        LALOC   0       ;adresse de tot au sommet de la pile
        LVLOCI  0       ;valeur de tot au sommet de la pile
        LVLOCI  1       ;valeur de i au sommet de la pile
        PLUSI           ;addition des 2 elements au sommet de la pile
        SVALI           ;affectation du resultat a tot
        REMVEI          ;enlever l'element au sommet de la pile


        new expression at line 12       ;expression i++

        LALOC   1       ;adresse de i au sommet de la pile
        INCAFT  1       ;increment i++
        REMVEI          ;enlever l'element au sommet de la pile
        JUMP    L2      ;branchement a L2


        RETRNI  2       ;restitution de l'environnement initial

TA      :0
TE      :0

TA      :0
TE      :0
```

III.2 Phase de traduction en code CALM

Comme nous l'avons déjà dit,cette phase de traduction a pour but de traduire les programmes VAC obtenus lors de la phase de pré-compilation en code assembleur universel CALM.Pour réaliser cette phase,nous avons choisi d'écrire des programmes SNOBOL qui assurent la traduction.

Nous allons tout d'abord expliquer les raisons qui nous ont amenés à choisir SNOBOL 3.Nous décrivons ensuite chacune des deux passes de la traduction:la première qui assure la traduction et demande au départ la définition des algorithmes de traduction pour chaque instruction VAC; la seconde permet d'optimiser le code CALM obtenu,par l'utilisation de sous-programmes et par la suppression d'instructions inutiles.Enfin,nous donnons un exemple de traduction pour quelques instructions VAC pour le Motorola 6500. Nous fournissons également un exemple complet de programme.

III.2I Raisons du choix de SNOBOL 3

Rappelons que SNOBOL 3 permet une manipulation aisée des chaînes de caractères.Il permet notamment de générer assez facilement du code symbolique en fonction de la reconnaissance d'une chaîne de caractères.C'est cet aspect qui nous intéresse au niveau de la phase de traduction.

Avant de choisir SNOBOL 3,nous avons réalisé un certain nombre d'essais pour déterminer quels étaient les problèmes que nous rencontrerions.Nous avons tout d'abord réalisé un essai de traduction à l'aide de l'éditeur de UNIX. Cette solution ne pouvait évidemment pas être retenue,car trop lente,mais elle nous a permis de mieux cerner le problème.Nous avons ensuite étudié les possibilités d'un macro-générateur -STAGE 2.Mais il nous a semblé trop complexe pour le genre de problème que nous devions résoudre.Nous avons ensuite étudié les possibilités et réalisé des essais

avec un autre macro-générateur -M6.M6 a été mis au point et implémenté en Fortran IV par M. Mc Ilroy et R. Morris des Bell Telephone Laboratories.Il a été compilé et exécuté sur GE-635,IBM 360/65,CDC 6600,Univac II08,PDP-I0,PDP II/45 et SIGMA 7.Ce macro-générateur aurait pu être retenu mais nous ne voulions pas,à l'époque,modifier le précompilateur VAC,or c'était nécessaire pour utiliser M6 de façon efficace.

Nous avons également discuté la possibilité d'écrire un macro-générateur propre au problème que nous devions résoudre.Mais par manque de temps,nous avons rejeté cette solution.

Nous avons donc retenu SNOBOL 3 dont le gros défaut est sa lenteur du fait qu'il travaille en mode interpréteur. Mais n'oublions pas que le but essentiel de cette étude était avant tout de découvrir un certain nombre de problèmes liés à la construction d'un cross-compilateur paramétrable et pas d'en optimiser les performances. (Certaines améliorations possibles seront citées dans les conclusions)

III.22 Description de la phase de traduction

III.22I Définition des algorithmes des instructions VAC

Le premier travail à réaliser est de traduire chaque instruction VAC en code CALM et donc d'écrire pour chaque instruction VAC l'algorithme CALM qui lui correspond.

Ces algorithmes peuvent être écrits indépendamment des techniques employées pour réaliser la traduction en CALM.

Nous allons donner ici un exemple d'algorithme.L'ensemble des algorithmes se trouve en annexe (voir annexe 3).

Comme exemple d'algorithme,nous avons choisi l'instruction VAC: PLUSI.Cette instruction effectue la somme des deux entiers se trouvant au sommet de la pile.Elle retire les deux opérandes du sommet de la pile et y place le résultat.L'algorithme pour le Motorola 6800 est le suivant:

```
POP B          ;extrais le byte au sommet de la pile (byte le plus significatif de l'opérande 2)
                dans l'accumulateur B.
POP A          ;extrais le byte au sommet de la pile (byte le moins significatif de l'opérande 2)
                dans l'accumulateur A.
INC IX,SP      ;le registre d'index IX pointe sur la première opérande (IX=SP+I  SP est le stack
                pointer).
ADD A,(IX)+I;additionne les bytes les moins significatifs des 2 opérandes dans l'accumulateur
                A.
ADDC B,(IX)+0
               ;additionne avec carry les bytes les plus significatifs des 2 opérandes dans l'accumulateur B.
LOAD (IX)+I,A
LOAD (IX)+0,B
               ;sauve le résultat de l'addition sur la pile
```

Donnons un autre exemple d'algorithme:l'instruction EQI.Rappelons que cette instruction compare les 2 éléments au sommet de la pile et les remplace par I s'ils sont égaux, par 0 sinon.

L'algorithme est le suivant:

```
POP B          ;extrais le byte au sommet de la pile (byte le plus significatif de l'opérande 2)
```

```
                dans l'accumulateur B.
POP A           ;extrais le byte au sommet de la pile (by-
                te le moins significatif de l'opérande 2)
                dans l'accumulateur A.
INC IX,SP       ;IX pointe sur l'opérande I.
COMP B,(IX)+0
                ;compare les bytes les plus significatifs
                des 2 opérandes.
JUMP,NE .+8     ;branche si != les opérandes sont diffé-
                rentes.
LOAD A,#I       ;A=I.
COMP A,(IX)+I
                ;compare les bytes les moins significatifs
                des 2 opérandes.
JUMP,EQ .+3     ;branche si == les opérandes sont iden-
                tiques.
CLR A           ;A=0.
CLR B           ;B=0.
LOAD (IX)+I,A
LOAD (IX)+0,B
                ;sauve le résultat au sommet de la pile.
```

III.222 Description de la première passe

A partir du moment où tous les algorithmes CALM sont définis,nous possédons pour chaque instruction VAC l'algorithme CALM qui lui correspond.

Il suffit alors d'avoir un programme qui lit le fichier VAC ligne par ligne.Ce programme doit pouvoir reconnaître l'instruction VAC se trouvant sur chaque ligne input et la remplacer par l'algorithme CALM qui lui correspond.

Le programme réalise les fonctions suivantes:

- lecture ligne par ligne du fichier VAC
- reconnaissance de la ligne lue
- en fonction de cette reconnaissance, génération de l'algorithme CALM qui correspond sur le fichier output

N.B. Nous verrons dans le paragraphe III.223 que le texte généré ne correspond pas toujours à l'algorithme CALM correspondant à une instruction VAC; cela pour des raisons d'optimisation du code généré.

Le programme complet se trouve en annexe. (voir annexe 2.d:Motorola.I) Nous allons l'illustrer ici pour les deux instructions PLUSI et EQI dont nous venons de donner les algorithmes CALM.

```
goI  input   =   syspit          /f(end)

**lecture d'une ligne du fichier VAC dans la zone input; si
**le fichier est vide branchement à end

     input *//8* 'PLUSI'         /s(plusi)

**test de la zone input. Si elle contient '        PLUSI'
**branchement à plusi

     input *//8* 'EQI'           /s(eqi)f(goI)

**idem mais pour EQI  en cas d'échec branchement à goI
```

```
plusi   syspot = 'POP B'
        syspot = 'POP A'
        syspot = 'INC IX,SP'
        syspot = 'ADD A,(IX)+I'
        syspot = 'ADDC B,(IX)+0'
        syspot = 'LOAD (IX)+I,A'
        syspot = 'LOAD (IX)+0,B'      /(goI)
```

**génère ces instructions CALM dans le fichier output et bran-
**che à goI.

```
eqi     syspot = 'POP B'
        syspot = 'POP A'
        syspot = 'INC IX,SP'
        syspot = 'COMP B,(IX)+0'
        syspot = 'JUMP,NE .+8'
        syspot = 'LOAD A, I'
        syspot = 'COMP A,(IX)+I'
        syspot = 'JUMP,EQ .+3'
        syspot = 'CLR A'
        syspot = 'CLR B'
        syspot = 'LOAD (IX)+I,A'
        syspot = 'LOAD (IX)+0,B'      /(goI)
```

**idem que pour plusi

```
end     syspot = ''
```

**fin du programme

III.223 Description de la seconde passe

Cette seconde passe de la phase de traduction a pour but d'optimiser le code CALM.Elle permet de supprimer certaines séquences d'instructions inutiles ou de les remplacer par des séquences plus courtes.Elle permet aussi d'obtenir un code plus concis grâce à l'utilisation de sous-routines.Elle génère également les instructions d'initialisation et de terminaison du programme.

III.223I Séquence d'initialisation et de terminaison du programme

La deuxième passe réalise la génération des instructions d'initialisation et de terminaison du programme,y compris la réservation de zones de travail.

Les fonctions réalisées par cette partie du programme sont les suivantes:

- réservation des variables de travail et initialisation de ces variables.Ces variables sont pour le motorola 6800:SH-SL,SSH-SSL,SSSH-SSSL,SSSSH-SSSSL,LH-LL (adresse de base locale).
- initialisation du stack pointer et de l'adresse de base locale.
- branchement à l'étiquette MAIN qui représente le début réel du programme.
- arrêt du programme.

III.2232 Suppression des séquences inutiles

Il apparaît,lors de l'étude du texte généré par la première passe,qu'un certain nombre de séquences d'instruc-

tion sont inutiles ou peuvent être réduites.La seconde passe permet de reconnaître ces séquences et de les supprimer.

Pour réaliser cette fonction,il est nécessaire d'utiliser un certain nombre de buffers inputs,au minimum égal au nombre d'instructions de la séquence la plus longue à reconnaître.Il faut alors tester ces différents buffers afin de voir si une telle séquence s'y trouve.Si c'est le cas, on ne recopie sur le fichier output que les instructions necessaires.

Afin d'illustrer cette fonction,voyons les séquences inutiles qui ont été dégagées pour le Motorola 6800.Nous expliquons ensuite la partie de programme qui réalise cette fonction.

- première séquence

```
PUSH A
PUSH B      ;sauve A et B au sommet de la pile
POP B
POP A       ;extrais les 2 bytes au sommet de la pile
```

Ces quatre instructions peuvent évidemment être supprimées lorsqu'elles se suivent.

- deuxième séquence

```
LOAD (IX)+I,A
LOAD (IX)+0,B  ;sauve A et B au sommet de la pile
POP B
POP A          ;extrais les 2 bytes au sommet
                de la pile
```

Cette séquence de 4 instructions peut être remplacée par:

```
INC SP
INC SP
```

En effet,cette séquence réalise les mêmes opérations que la première séquence mais il faut tenir compte du fait que l'instruction POP incrémente automatiquement le stack pointer.

Voyons maintenant la partie de programme qui réalise cette fonction:un buffer de 4 lignes est nécessaire puisque le nombre maximum d'instructions à reconnaître est de 4.

```
readI   input1 = syspit
        input2 = syspit
        input3 = syspit
        input4 = syspit

**lecture de 4 lignes du fichier input

test1   input1 ** 'PUSH A'      /f(test2)
        input2 ** 'PUSH B'      /f(test2)
        input3 ** 'POP B'       /f(test2)
        input4 ** 'POP A'       /s(read1)

**cette suite d'instructions signifie que si:
**      input1 contient 'PUSH A'
**      input2 contient 'PUSH B'
**      input3 contient 'POP B'
**      input4 contient 'POP A' ,
**ces 4 instructions ne sont pas recopiées dans le fichier
**output,si ce n'est pas le cas branchement à test2.
```

```
test2   inputI ** 'LOAD (IX)+I,A'  /f(chang)
        input2 ** 'LOAD (IX)+0,B'  /f(chang)
        input3 ** 'POP B'          /f(chang)
        input4 ** 'POP A'          /f(chang)
        syspot  = 'INC SP'
        syspot  = 'INC SP'         /(readI)
```

**même mécanisme que pour testI mais génération de 'INC SP'
** et 'INC SP' dans le fichier output.

chang suite du programme

III.2233 Utilisation de sous-routines

Nous avons dit dans le paragraphe III.222 que la première passe de la phase de traduction remplaçait chaque instruction V.C par l'algorithme CALM qui lui correspond. Ceci n'est pas toujours vrai. Lorsque l'algorithme CALM comprend un trop grand nombre d'instructions, nous avons essayé dans la mesure du possible, d'en faire une sous-routine qui ne sera générée qu'une fois et de ne générer dans le texte du programme que les instructions CALM absolument nécessaires - appel aux sous-routines et instructions de manipulation de la pile ou de modification du stack pointer.

La deuxième passe de la phase de traduction permet de reconnaître les sous-routines nécessaires et de les générer.

A titre d'exemple reprenons l'exemple donné en III.221 pour l'instruction EQI pour le Motorola 6800.

Algorithme	instructions générées première passe	instructions générées deuxième passe
POP B POP A INC IX,SP COMP B,(IX)+0 JUMP,NE .+8 LOAD A,#I COMP A,(IX)+I JUMP,EQ .+3 CLR A CLR B LOAD (IX)+I,A LOAD (IX)+0,B	POP B POP A INC IX,SP CALL EQI	EQI: COMP B,(IX)+0 JUMP,NE .+8 LOAD A,#I COMP A,(IX)+I JUMP,EQ .+3 CLR A CLR B LOAD (IX)+I,A LOAD (IX)+0,B RET

L'avantage de cette technique est évident. Si l'instruction EQI doit être générée plusieurs fois dans le même programme, seules les 4 instructions générées lors de la première passe le seront chaque fois, la sous-routine ne l'étant qu'une fois.

Il faut noter que les instructions de manipulation de la pile et de modification du stack pointer ne peuvent généralement pas se trouver dans la sous-routine car les instructions CALL et RET utilisent également la pile pour retenir l'adresse de retour après exécution de la sous-routine.

Voyons maintenant la partie du programme qui realise cette fonction. Elle est réalisée grâce à deux fonctions: appel.fnc et test.fnc.

- fonction appel.fnc

Cette fonction permet de tester quelles sont les routines à générer.

Une variable globale initialisée à 0 est utilisée pour chaque sous-routine.Si une ligne du fichier output contient un appel à une sous-routine,la variable qui lui correspond est positionnée à I.

- exemple pour EQI

```
            eqi = '0'           ;initialisation de eqi
            input = syspit      ;lecture dans input
            appel.fnc(input)    ;appel de la fonction
            syspot = input      ;écriture de input

define      appel.fnc(input)
appelI      input      'EQI'    /f(appel2)
            eqi = 'I'           /(return)
appel2      suite de la fonction
```

- fonction test.fnc

Cette fonction génère le texte de toutes les sous-routines dont la variable globale correspondante est positionnée à I.Elle est appelée en fin de programme lorsque tout le fichier input a été lu.

- exemple pour EQI

```
define    test.fnc(par)

testI     eqi 'I'              /f(test2)
          syspot = 'EQI:'
          syspot = 'COMP B,(IX)+0'
          syspot = 'JUMP,NE .+8'
          syspot = 'LOAD A,#I'
          syspot = 'COMP A,(IX)+I'
          syspot = 'JUMP,EQ .+3'
          syspot = 'CLR A'
          syspot = 'CLR B'
          syspot = 'LOAD (IX)+I,A'
          syspot = 'LOAD (IX)+0,B'
          syspot = 'RET'
test2     suite de la fonction
```

N.B. Le programme réalisant la seconde passe se trouve en annexe (voir annexe 2.d:Motorola.2).

III.23 Exemple de traduction

Reprenons le programme que nous avons présentés en III.I32 et voyons les résultats après la première et la seconde passe de la phase de traduction.

```
. text

instruction     MAIN:    ENTRY

AIN: LOAD A,LL
OAD B,LH
USH A
USH B

DB      i=-1 + NLOC
DB      tot=-2 + NLOC

instruction     SAVE     2

OAD LH,SP
OAD B,LH
OAD A,LL
UB A,#-1+2*2
UBC B,#0
OAD LH,B
OAD LL,A
OAD SP,LH
EC SP

DB      new expression at line 11


instruction     LALOC    0

OAD A,LL
OAD B,LH
OAD IX,#2*0
OAD SH,IX
OD A,SL
DC B,SH
USH A
USH B

instruction     LVCONI   0

OAD IX,#0
OAD SH,IX
OAD A,SL
OAD B,SH
PUSH A
PUSH B

instruction     SVALI

OP B
OP A
NC IX,SP
NC SP
NC SP
OAD IX,(IX)+0
OAD (IX)+1,A
OAD (IX)+0,B
PUSH A
PUSH B
```

```
;instruction    REMVEI

;POP B ,
;POP A

;DB     new expression at line 12


;instruction    LALOC   1

LOAD A,LL
LOAD B,LH
LOAD IX,#2*1
LOAD SH,IX
ADD A,SL
ADDC B,SH
PUSH A
PUSH B

;instruction    LVCONI  0

LOAD IX,#0
LOAD SH,IX
LOAD A,SL
LOAD B,SH
PUSH A
PUSH B

;instruction    SVALI

POP B
POP A
INC IX,SP
INC SP
INC SP
LOAD IX,(IX)+0
LOAD (IX)+1,A
LOAD (IX)+0,B
PUSH A
PUSH B

;instruction    REMVEI

POP B
POP A
2:

;DB     new expression at line 12


;instruction    LVLOCI  1

LOAD IX,LH
LOAD B,(IX)+2*1
LOAD A,(IX)+1+2*1
PUSH A
PUSH B

;instruction    LVCONI  50
```

```
LOAD IX,#50
LOAD SH,IX
LOAD A,SL
LOAD B,SH
*PUSH A
*PUSH B

; instruction     JGT     L3

*POP B
*POP A
INC IX,SP
INC SP
INC SP
COMP B,(IX)+0
JUMP,LT .+8
JUMP,GT .+9
COMP A,(IX)+1
JUMP,GE .+5
JUMP L3

; DB      new expression at line 13


; instruction     LALOC   0

LOAD A,LL
LOAD B,LH
LOAD IX,#2*0
LOAD SH,IX
ADD A,SL
ADDC B,SH
PUSH A
PUSH B

; instruction     LVLOCI  0

LOAD IX,LH
LOAD B,(IX)+2*0
LOAD A,(IX)+1+2*0
PUSH A
PUSH B

; instruction     LVLOCI  1

LOAD IX,LH
LOAD B,(IX)+2*1
LOAD A,(IX)+1+2*1
PUSH A
PUSH B

; instruction     PLUSI

POP B
POP A
INC IX,SP
ADD A,(IX)+1
ADDC B,(IX)+0
*LOAD (IX)+1,A
```

```
**LOAD (IX)+0,B

; instruction     SVALI

**POP B
**POP A
INC IX,SP
INC SP
INC SP
LOAD IX,(IX)+0
LOAD (IX)+1,A
LOAD (IX)+0,B
*PUSH A
*PUSH B

; instruction     REMVEI

*POP B
*POP A
_4:

_DB      new expression at line 12


 instruction     LALOC   1

LOAD A,LL
LOAD B,LH
LOAD IX,#2*1
LOAD SH,IX
DD A,SL
DDC B,SH
PUSH A
PUSH B

 instruction     INCAFT  1

POP B
POP A
OAD SH,B
OAD B,#1
ALL INCAFT
PUSH A
PUSH B

instruction     REMVEI

POP B
POP A

instruction     JUMP    L2

JMP L2

3:
1:

instruction     RETRNI  2

OAD IX,LH
```

```
LOAD IX,(IX)+2*2
LOAD SSH,IX
LOAD IX,LH
LOAD IX,(IX)+2+2*2
LOAD SSSH,IX
LOAD IX,LH
LOAD IX,(IX)+4+2*2
LOAD A,#2
JUMP RETRNI

*DATA  :0
*BYTE  :0

*DATA  :0
*BYTE  :0
```

Les instructions qui commencent par * sont des instructions inutiles qui seront supprimees lors de la phase d'optimisation.

Les instructions qui commencent par ** sont des instructions qui seront reduites lors de la phase d'optimisation.

```
sequence d'initialisation et de terminaison

RDX 10.
H:  .BYTE 0
L:  .BYTE 0
H:  .BYTE 0
L:  .BYTE 0
SH:  .BYTE 0
SL:  .BYTE 0
SSH:  .BYTE 0
SSL:  .BYTE 0
SSSH:  .BYTE 0
SSSL:  .BYTE 0
DRINIT:  .WORD 1000
EBUT:  .LOC DEBUT
OAD SP,ADRINIT
OAD IX,ADRINIT
NC IX
OAD LH,IX
LR A
USH A
USH A
USH A
USH A
ALL MAIN
IN: WAIT

.text

instruction     MAIN:    ENTRY

AIN: LOAD A,LL
OAD B,LH
USH A
USH B

DB      i=-1 + NLOC
DB      tot=-2 + NLOC

instruction     SAVE    2

OAD LH,SP
OAD B,LH
OAD A,LL
UB A,#-1+2*2
UBC B,#0
OAD LH,B
OAD LL,A
OAD SP,LH
EC SP

DB      new expression at line 11


instruction     LALOC    0

OAD A,LL
OAD B,LH
OAD IX,#2*0
```

```
LOAD SH, IX
ADD A, SL
ADDC B, SH
PUSH A
PUSH B

;instruction     LVCONI   0

LOAD IX, #0
LOAD SH, IX
LOAD A, SL
LOAD B, SH

;instruction     SVALI

INC IX, SP
INC SP
INC SP
LOAD IX, (IX)+0
LOAD (IX)+1, A
LOAD (IX)+0, B


;instruction     REMVEI   suprimee

DB      new expression at line 12


;instruction     LALOC    1

OAD A, LL
OAD B, LH
OAD IX, #2*1
OAD SH, IX
DD A, SL
DDC B, SH
USH A
USH B

;instruction     LVCONI   0

OAD IX, #0
OAD SH, IX
OAD A, SL
OAD B, SH

;instruction     SVALI

NC IX, SP
NC SP
NC SP
OAD IX, (IX)+0
OAD (IX)+1, A
OAD (IX)+0, B

instruction     REMVEI   supprimee

2:

)B      new expression at line 12
```

```
instruction     LVLOCI   1

OAD IX,LH
OAD B,(IX)+2*1
OAD A,(IX)+1+2*1
USH A
USH B

instruction     LVCONI   50

OAD IX,#50
OAD SH,IX
OAD A,SL
OAD B,SH

instruction     JGT      L3

NC IX,SP
NC SP
NC SP
OMP B,(IX)+0
UMP,LT .+8
UMP,GT .+9
OMP A,(IX)+1
UMP,GE .+5
UMP L3

DB      new expression at line 13

instruction     LALOC    0

OAD A,LL
OAD B,LH
OAD IX,#2*0
OAD SH,IX
D A,SL
DC B,SH
SH A
SH B

instruction     LVLOCI   0

AD IX,LH
AD B,(IX)+2*0
AD A,(IX)+1+2*0
SH A
SH B

instruction     LVLOCI   1

AD IX,LH
AD B,(IX)+2*1
AD A,(IX)+1+2*1

instruction     PLUSI

C IX,SP
```

```
ADD A,(IX)+1
ADDC B,(IX)+0

** INC SP
** INC SP


; instruction    SVALI

INC IX,SP
INC SP
INC SP
LOAD IX,(IX)+0
LOAD (IX)+1,A
LOAD (IX)+0,B

 instruction    REMVEI  supprimee

_4:

 DB     new expression at line 12


 instruction    LALOC   1

LOAD A,LL
LOAD B,LH
LOAD IX,#2*1
LOAD SH,IX
ADD A,SL
ADDC B,SH

 instruction    INCAFT  1

LOAD SH,B
LOAD B,#1
CALL INCAFT

 instruction    REMVEI  supprimee


 instruction    JUMP    L2

JUMP L2

3:
1:

 instruction    RETRNI  2

LOAD IX,LH
LOAD IX,(IX)+2*2
LOAD SSH,IX
LOAD IX,LH
LOAD IX,(IX)+2+2*2
LOAD SSSH,IX
LOAD IX,LH
LOAD IX,(IX)+4+2*2
LOAD A,#2
```

```
JUMP RETRNI

 *DATA  :0
 *BYTE  :0

 *DATA  :0
 *BYTE  :0


routine RETRNI


ETRNI:
OAD SH,IX
LC SL
LC A
DD A,#8
DD A,SL
LR B
DD A,LL
DDC B,LH
OAD SL,A
OAD SH,B
LR A
LR B
OAD SSSSH,SP
OAD IX,SSSSH
NC IX
OMP IX,LH
UMP,EQ .+4
OP A
OP B
OAD IX,SSH
OAD LH,IX
OAD SP,SH
EC SP
USH B
USH A
OAD IX,SSSH
UMP (IX)+0


routine INCAFT


NCAFT:
OAD SL,A
OAD IX,SH
OAD A,(IX)+1
USH A
DD A,B
OAD (IX)+1,A
OAD B,(IX)+0
USH B
DDC B,#0
OAD (IX)+0,B
OP A
OP B
ET
```

END

Les instructions precedees de ** correspondent a des
ontractions de texte:

```
LOAD (IX)+1,A
LOAD (IX)+0,B
POP B
POP A
```

st remplace par:

```
INC SP
INC SP
```

On remarque notamment que la séquence - PUSH A;PUSH B;POP B;POP A - se rencontre 9 fois.Il y a donc un gain de 36 instructions.L'autre séquence - LOAD (IX)+I,A; LOAD (IX)+0,B;POP B;POP A - est rencontée une fois.On gagne donc 2 instructions.Au total,il y a 38 instructions en moins sur un total de 151 instructions.Le gain est donc appréciable puisqu'il représente plus de 20%.

Au niveau du nombre de bytes générés,on gagne 36 bytes pour la première séquence et 4 pour la seconde,soit 40 bytes sur un total de 251.Le gain est de l'ordre de 15%.

Il semble que ces chiffres peuvent être généralisés: on gagne donc plus ou moins 20% dans le nombre d'instructions et 15% dans le nombre de bytes.

III.3 Phase de cross-assemblage

Rappelons que la phase de cross-assemblage a pour but de transformer le programme CALM obtenu lors de la phase de traduction en un programme exécutable.Le cross-assemblage est paramétrable en ce sens qu'il permet d'obtenir des programmes exécutables pour différents processeurs.Il suffit pour cela de fournir au cross-assembleur une description correspondant au processeur sur lequel on veut exécuter le programme.

Rappelons également que le cross-assembleur utilisé est le cross-assembleur UNIASS,écrit en Pascal par M. Schmit de l'Ecole Polytechnique Fédérale de Lausanne.

Dans ce paragraphe,nous allons présenter les règles de construction de la description permettant de paramétrer le cross-assembleur pour un processeur donné.Nous appliquons ensuite ces règles pour quelques instructions du Motorola 6800.La description complète pour le Motorola 6800 se trouve en annexe (voir annexe 2.c).Nous donnons également un exemple concret de cross-assemblage d'un programme pour le Motorola 6800.

III.3I Règles de construction de la description pour UNIASS

La description à fournir à UNIASS n'est pas une description de l'architecture de la machine mais une description formelle du langage assembleur dans laquelle la machine doit être programmée.Il s'agit donc dans notre cas d'une description des instructions CALM utilisées par le processeur pour lequel on réalise la description.

UNIASS dispose d'un certain nombre d'outils pour aider l'utilisateur à réaliser une description;il s'agit d' un ensemble de pseudo-instructions de description et d'un ensemble de variables.

Ce sont ces variables et ces pseudo-instructions que nous allons décrire maintenant.

III.3II Définition des variables UNIASS

Toutes les variables qui vont être définies peuvent être utilisées explicitement dans le texte de la description.

- variable V

Il s'agit de la variable la plus importante,elle contient la dernière valeur qui a été évaluée par l'assembleur en analysant une ligne du progamme source.Par exemple,après avoir évalué une expression,sa valeur est placée dans la variable V.La description devra donc indiquer ce qui doit être fait du contenu de V.

- variables Vi (i=I,...,9)

Ces 9 variables permettent de stocker temporairement des valeurs.

- variables Gi (i=I,...,9)

Ces 9 variables permettent de stocker des informations permanentes.

- bytes Bi (i=I,...,9)

Ces 9 variables permettent la génération de code en plaçant des valeurs dans les Bi.A la fin d'une instruction, ces variables sont transférées dans le fichier output si des valeurs leur ont été affectées.Ne disposant que de 9 variables Bi,il n'est pas possible de générer plus de 9 bytes par instruction.

- P-counter PC

Le P-counter est accessible au programmeur.Il s'agit de l'adresse suivante où on doit générer du code.

III.3I2 Description des pseudo-instructions

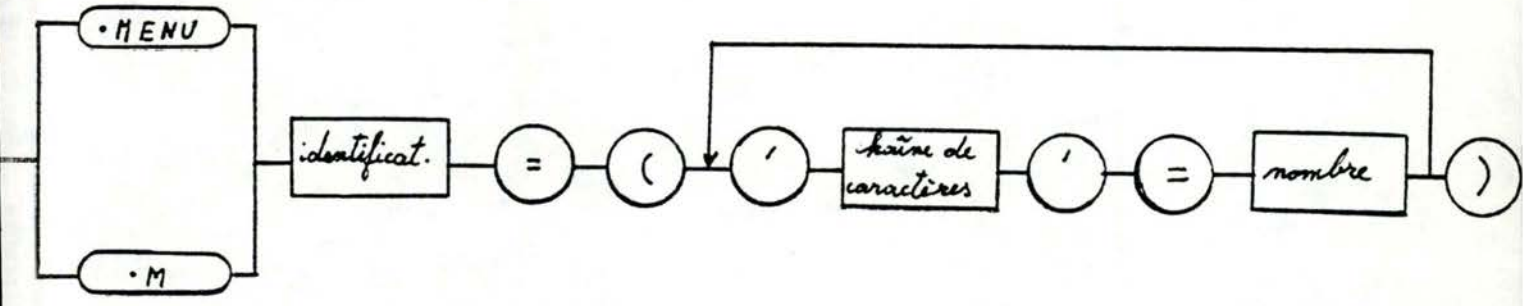
Les pseudo-instructions sont au nombre de cinq:

.MENU
.CODE
.TEST
.MAXI
.INSTR

- .MENU

La pseudo-instruction .MENU est utilisée pour grouper sous le même nom un certain nombre de paramètres qui peuvent apparaître à la même place dans une instruction,en assignant une valeur à chaque paramètre.Cette valeur sera utilisée pour générer le code.L'assembleur crée une liste des éléments du menu.La recherche dans cette liste est séquentielle.Lors d'une recherche fructueuse dans la liste,la valeur correspondante est placée dans la variable V et peut donc être utilisée pour générer le code.

syntaxe



exemple

Les instructions PUSH et POP pour le Motorola 6800 sont codées de la façon suivante:

62+	POP p	p	0	A
66+	PUSH p		I	B

Une instruction .MENU peut être définie pour exprimer quel registre (A ou B) est utilisé dans l'instruction.

.MENU PPAB = ('A' = 0,'B' = I)

Lorsque l'assembleur exécute cette instruction,s'il reconnaît dans la ligne qu'il traite une des deux chaînes de caractères 'A' ou 'B',il place dans la variable V la valeur qui est associée à la chaîne de caractères reconnue.

Nous verrons plus loin comment cette valeur sera utilisée

Une autre pseudo-instruction .MENU peut être utilisée pour ce même exemple:

.MENU PP = ('PUSH' = 66,'POP' = 62)

Ici,V contiendra 66 ou 62 suivant que l'instruction

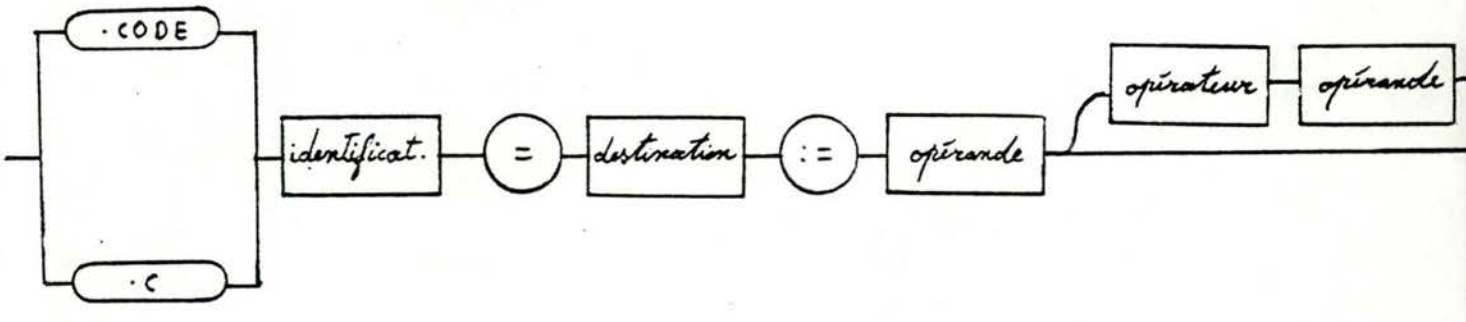
traitée est respectivement PUSH ou POP.

- .CODE

La pseudo-instruction .CODE permet d'effectuer des opérations arithmétiques et logiques sur les variables du cross-assembleur:V,Vi,Gi,Bi ou PC.Des valeurs quelconques peuvent être affectées à chacune de ces variables.C'est un moyen utilisé pour décrire la génération de code (en affectant des valeurs aux variables Bi).Nous verrons plus loin que des valeurs peuvent aussi être affectées aux variables Bi directement dans la description d'une instruction.

Il existe une pseudo-instruction .CODE prédéfinie ZEROi (i=I,...,9) qui permet d'annuler la valeur placée dans Bi quand il apparaît que celle-ci n'est plus nécessaire.ZEROi est différente d'une pseudo-instruction .CODE qui spécifie Bi:=0 car après ZEROi,Bi est mis à 0 mais en plus,le byte Bi n'est pas pris en considération pour générer le code.

syntaxe



```
destination ::= V,Vi,Gi,PC,Bi
operande    ::= V,Vi,Gi,PC,Bi ou nombre
operateur   ::=| &   AND logique
               | !   OR logique
               | +   addition
               | -   soustraction
               | *   multiplication
               | ->  shift à droite
               | <-  shift à gauche
```

exemple

Des pseudo-instructions .CODE peuvent être utilisées pour transformer un nombre 16 bits en 2 bytes et de les placer dans les bytes 2 et 3 d'une instruction.On obtient:

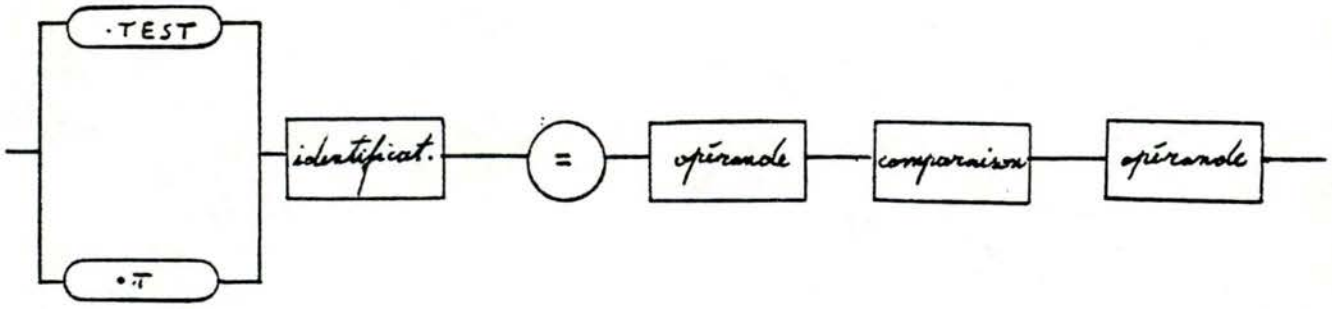
```
.CODE LOWB = VI:=V&377          ;VI:=low byte
.CODE HBI  = V2:=V&I77400
.CODE HB2  = V3:=V2- 8          ;V3:=high byte
.CODE B2V3 = B2:=V3
.CODE B3VI = B3:=VI
```

Les bytes B2 et B3 contiennent après exécution de cette séquence d'instructions les bytes provenant du nombre 16 bits qui se trouvait dans la variable V.

- .TEST

La pseudo-instruction .TEST permet de comparer les variables V,Vi,Gi,Bi,PC;ce qui permet à l'assembleur de prendre certaines décisions en fonction du résultat du test.

syntaxe



comparaison ::= <,>,=,<=,>=,<> (!=)

exemple

```
.TEST A = V=0
.TEST B = V=I00
```

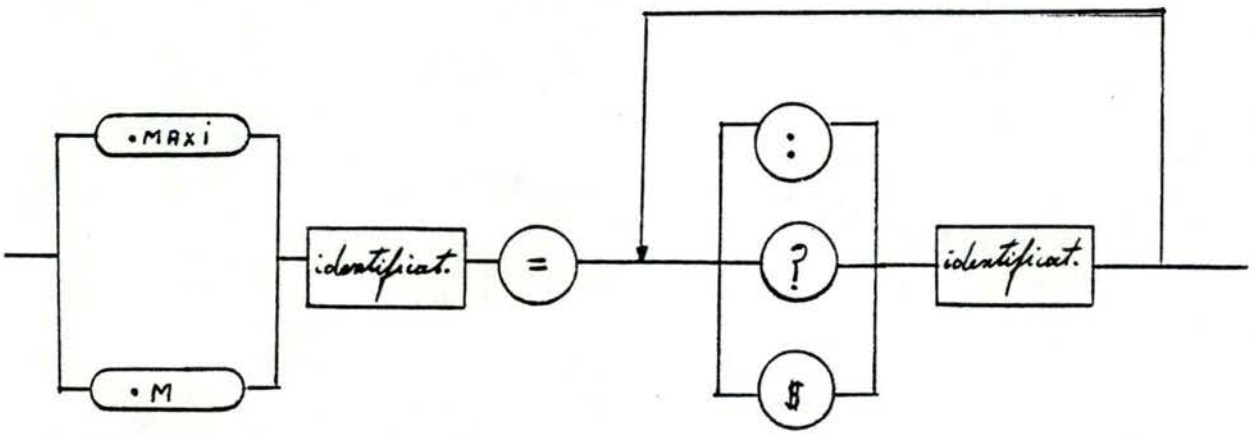
Ces deux pseudo-instructions permettent de tester la valeur de V,la première si V vaut 0,l'autre si V vaut I00.

- .MAXI

La pseudo-instruction .MAXI est utilisée pour grouper sous un même nom une séquence de pseudo-instructions .CODE et .TEST qui apparaissent souvent ensembles.Une instruction .MAXI peut aussi contenir une autre instruction .MAXI.Pour pouvoir différencier les paramètres (.CODE,.TEST,.MAXI),un nom de test est précédé par un point d'interrogation (?), un nom de code est précédé par le caractère':' et un nom de maxi par un dollar (**$**).

Si une pseudo-instruction contient plusieurs tests,le résultat global est un AND logique de tous les tests:.MAXI réussit seulement si tous les tests qu'il contient réussissent.

syntaxe



exemple

Si l'on reprend l'exemple pour transformer un nombre de bits en deux bytes, donné pour illustrer la pseudo-instruction .CODE, toutes ces pseudo-instructions peuvent être regroupées pour être exécutées ensembles par la pseudo-instruction:

.MAXI M2 = :LOWB, :HBI, :HB2, :B2V3, :B3VI

.INSTR

La pseudo-instruction .INSTR permet de décrire les instructions du langage assembleur. Chaque pseudo-instruction .INSTR crée une partie de l'arbre qui sera parcouru par l'assembleur pour analyser chaque instruction du programme source.

Les paramètres possibles pour cette pseudo-instruction sont:

- une chaîne de caractères à reconnaître. Cette chaîne de caractères est composée de maximum 6 caractères et est mise entre quotes. La valeur de V n'est pas modifiée

si l'assembleur reconnaît une telle chaîne de caractères.

exemples: 'LOAD' '(IX)+'

- le nom d'un menu mis entre parenthèses.L'assembleur essaie de reconnaître une des chaînes de caractères qui composent le menu;en cas de succès,la valeur associée à cette chaîne est placée dans la variable V.

exemples: (PP) (PPAB)

- la présence d'un séparateur dans la ligne input analysée sera représentée par un tiret (_)

- La reconnaissance d'une expression est décrite de la façon suivante: E(n) ou S(n). n représente le nombre maximum de bits de l'expression.E(n) signifie que l'expression est considerée comme non signée.S(n) signifie que l'expression est signée.

exemples: une adresse 16 bits est représentée par E(16.).L'assembleur vérifie que 0 <=expression >=177777 (valeur octale).

N.B. Le '.' signifie que la valeur est décimale

- le nom d'un code,test ou maxi précédé par le caractère spécial utilisé pour les distinguer,respectivement ':','?',#.

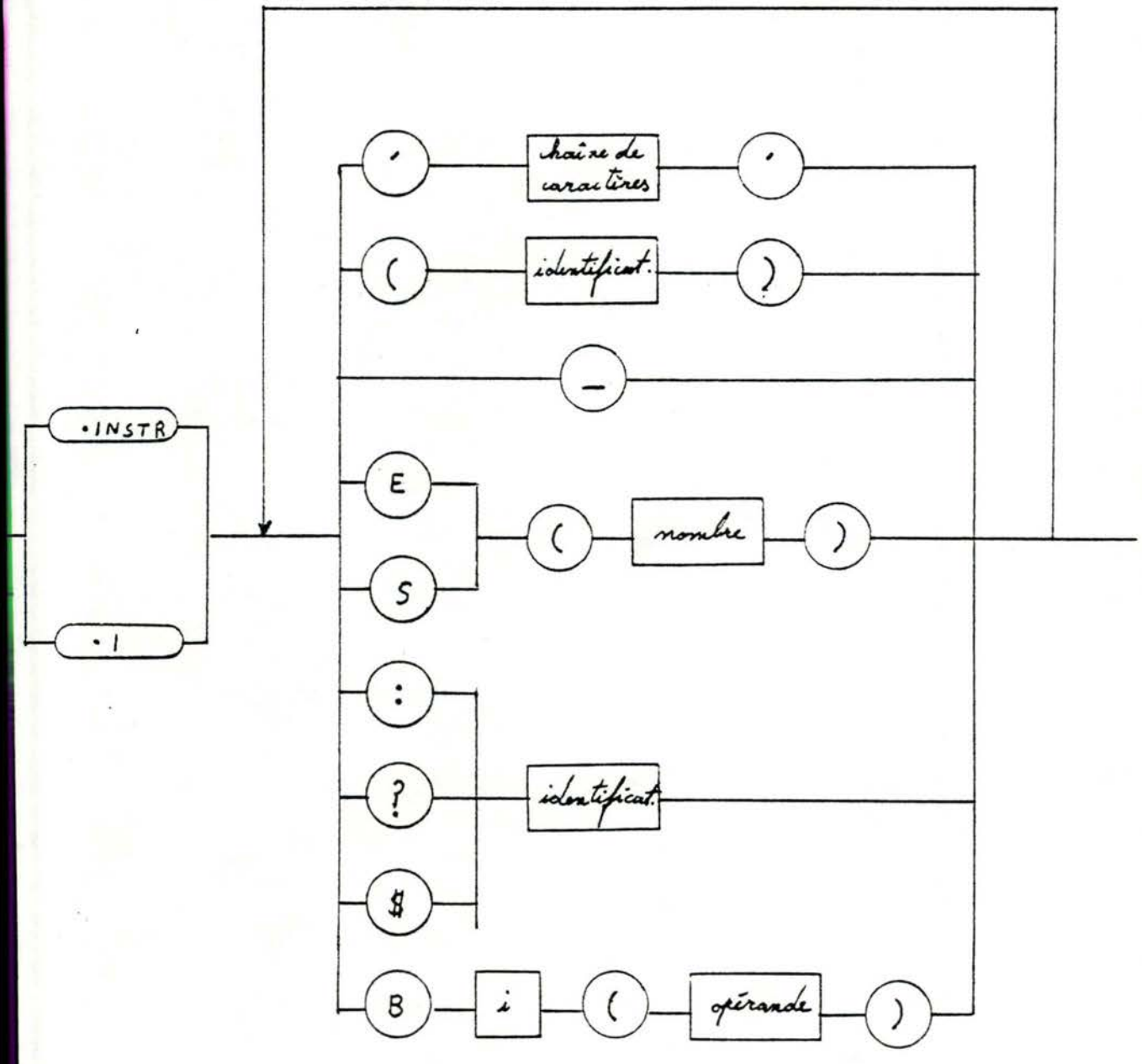
```
exemples: ?A          ;test
          #W2         ;maxi
          :LOWB       ;code
```

- la génération de code peut être décrite par le paramètre Bi(val) (i=1,...,9). i est le numéro du byte dans lequel la valeur 'val' doit être placée. val peut être un

nombre ou une des variables V,Vi,Gi,Bi,PC.La valeur est additionnée à la valeur précédente contenue dans Bi.Toutes les variables Bi sont remises à zéro au commencement de chaque ligne du programme source.

exemples: BI(370)
B2(V)

syntaxe



III.3I3 Exemples de description

Donnons quelques exemples d'application de ces règles pour quelques instructions du Motorola 6800

- LOAD F,A (F)- A dont le code objet est 006

```
.I 'LOAD'_'F'_'A'BI(6)
```

signifie que si l'assembleur rencontre une ligne input contenant dans l'ordre la chaîne de caractères 'LOAD' puis un séparateur suivi de 'F',d'un nouveau séparateur et de 'A' un byte contenant 6 sera généré.

- LOAD (IX)+N,R (avec R = A,B,IX ou SP)

```
.MENU R = ('A'=0,'B'=I00,'IX'=II0,'SP'=I0)
.I 'LOAD'_'(IX)+'E(8.)B2(V)_(R)BI(V)BI(247)
```

Le menu R permet de construire le code opératoire de l'instruction.Celui-ci vaut en effet

247+0 pour LOAD(IX)+N,A
247+I00 pour LOAD (IX)+N,B
247+II0 pour LOAD (IX)+N,IX
247+I0 pour LOAD (IX)+N,SP

L'instruction .I réalise respectivement les opérations suivantes:

. reconnaissance de 'LOAD'
. reconnaissance d'un séparateur
. reconnaissance de '(IX)+'
. reconnaissance d'une expression 8 bits non signée E(8.) et mise de la valeur de cette expression dans V

. place V dans le byte B2 (B2(V)) qui contient la valeur de l'expression E(8.)

. reconnaissance d'un séparateur

. exécution du menu R,s'il réussit place la valeur correspondante dans V

. place V dans BI (BI(V))

. ajoute 247 à BI (BI(247))

. si toutes ces opérations réussissent génération du code de l'instruction en plaçant BI et B2 dans le fichier output

Rappelons qu'on trouve en annexe la description complète du motorola 6800.(voir annexe2.c)

III.32 Exemple de cross-assemblage

Le programme utilisé est celui que nous utilisons depuis le début de ce chapitre.

```
                                .RDX 10.
000000    000                   LH:  .BYTE 0
000001    000                   LL:  .BYTE 0
000002    000                   SH:  .BYTE 0
000003    000                   SL:  .BYTE 0
000004    000                   SSH:  .BYTE 0
000005    000                   SSL:  .BYTE 0
000006    000                   SSSH:  .BYTE 0
000007    000                   SSSL:  .BYTE 0
000010    000                   SSSSH:  .BYTE 0
000011    000                   SSSSL:  .BYTE 0
000012    350 003               ADRINIT:  .WORD 1000
                                DEBUT:  .LOC DEBUT
000014    236 012               LOAD SP,ADRINIT
000016    336 012               LOAD IX,ADRINIT
000020    010                   INC IX
000021    337 000               LOAD LH,IX
000023    117                   CLR A
000024    066                   PUSH A
000025    066                   PUSH A
000026    066                   PUSH A
000027    066                   PUSH A
000030    275 000 034           CALL MAIN
000033    076                   FIN:  WAIT

                                ;.text
000034    226 001               MAIN:  LOAD A,LL
000036    326 000               LOAD B,LH
000040    066                   PUSH A
000041    067                   PUSH B
                                ;DB       i=-1 + NLOC
                                ;DB       tot=-2 + NLOC
000042    237 000               LOAD LH,SP
000044    326 000               LOAD B,LH
000046    226 001               LOAD A,LL
000050    200 003               SUB A,#-1+2*2
000052    302 000               SUBC B,#0
000054    327 000               LOAD LH,B
000056    227 001               LOAD LL,A
000060    236 000               LOAD SP,LH
000062    064                   DEC SP

                                ;DB       new expression at line 11

000063    226 001               LOAD A,LL
000065    326 000               LOAD B,LH
000067    316 000 000           LOAD IX,#2*0
000072    337 002               LOAD SH,IX
000074    233 003               ADD A,SL
000076    331 002               ADDC B,SH
000100    066                   PUSH A
000101    067                   PUSH B
000102    316 000 000           LOAD IX,#0
000105    337 002               LOAD SH,IX
000107    226 003               LOAD A,SL
000111    326 002               LOAD B,SH
000113    060                   INC IX,SP
000114    061                   INC SP
000115    061                   INC SP
000116    356 000               LOAD IX,(IX)+0
```

```
000120   247 001              LOAD (IX)+1,A
000122   347 000              LOAD (IX)+0,B

                              ;DB      new expression at line 12

000124   226 001              LOAD A,LL
000126   326 000              LOAD B,LH
000130   316 000 002          LOAD IX,#2*1
000133   337 002              LOAD SH,IX
000135   233 003              ADD A,SL
000137   331 002              ADDC B,SH
000141   066                  PUSH A
000142   067                  PUSH B
000143   316 000 000          LOAD IX,#0
000146   337 002              LOAD SH,IX
000150   226 003              LOAD A,SL
000152   326 002              LOAD B,SH
000154   060                  INC IX,SP
000155   061                  INC SP
000156   061                  INC SP
000157   356 000              LOAD IX,(IX)+0
000161   247 001              LOAD (IX)+1,A
000163   347 000              LOAD (IX)+0,B
                              L2:

                              ;DB      new expression at line 12

000165   336 000              LOAD IX,LH
000167   346 002              LOAD B,(IX)+2*1
000171   246 003              LOAD A,(IX)+1+2*1
000173   066                  PUSH A
000174   067                  PUSH B
000175   316 000 062          LOAD IX,#50
000200   337 002              LOAD SH,IX
000202   226 003              LOAD A,SL
000204   326 002              LOAD B,SH
000206   060                  INC IX,SP
000207   061                  INC SP
000210   061                  INC SP
000211   341 000              COMP B,(IX)+0
000213   055 006              JUMP,LT .+8
000215   056 007              JUMP,GT .+9
000217   241 001              COMP A,(IX)+1
000221   054 003              JUMP,GE .+5
000223   176 000 331          JUMP L3

                              ;DB      new expression at line 13

000226   226 001              LOAD A,LL
000230   326 000              LOAD B,LH
000232   316 000 000          LOAD IX,#2*0
000235   337 002              LOAD SH,IX
000237   233 003              ADD A,SL
000241   331 002              ADDC B,SH
000243   066                  PUSH A
000244   067                  PUSH B
000245   336 000              LOAD IX,LH
000247   346 000              LOAD B,(IX)+2*0
000251   246 001              LOAD A,(IX)+1+2*0
000253   066                  PUSH A
```

```
000254   067                    PUSH B
000255   336 000                LOAD IX,LH
000257   346 002                LOAD B,(IX)+2*1
000261   246 003                LOAD A,(IX)+1+2*1
000263   060                    INC IX,SP
000264   253 001                ADD A,(IX)+1
000266   351 000                ADDC B,(IX)+0
000270   061                    INC SP
000271   061                    INC SP
000272   060                    INC IX,SP
000273   061                    INC SP
000274   061                    INC SP
000275   356 000                LOAD IX,(IX)+0
000277   247 001                LOAD (IX)+1,A
000301   347 000                LOAD (IX)+0,B
                                L4:

                                ;DB      new expression at line 12

000303   226 001                LOAD A,LL
000305   326 000                LOAD B,LH
000307   316 000 002            LOAD IX,#2*1
000312   337 002                LOAD SH,IX
000314   233 003                ADD A,SL
000316   331 002                ADDC B,SH
000320   327 002                LOAD SH,B
000322   306 001                LOAD B,#1
000324   275 001 033            CALL INCAFT
000327   040 234                JUMP L2
                                L3:
                                L1:
000331   336 000                LOAD IX,LH
000333   356 004                LOAD IX,(IX)+2*2
000335   337 004                LOAD SSH,IX
000337   336 000                LOAD IX,LH
000341   356 006                LOAD IX,(IX)+2+2*2
000343   337 006                LOAD SSSH,IX
000345   336 000                LOAD IX,LH
000347   356 010                LOAD IX,(IX)+4+2*2
000351   206 002                LOAD A,#2
000353   176 000 356            JUMP RETRNI

                                ;*DATA   :0
                                ;*BYTE   :0

                                ;*DATA   :0
                                ;*BYTE   :0



                                RETRN1:
000356   337 002                LOAD SH,IX
000360   170 000 003            SLC SL
000363   110                    SLC A
000364   213 010                ADD A,#8
000366   233 003                ADD A,SL
000370   137                    CLR B
000371   233 001                ADD A,LL
000373   331 000                ADDC B,LH
```

```
000375   227 003              LOAD SL,A
000377   327 002              LOAD SH,B
000401   117                  CLR A
000402   137                  CLR B
000403   237 010              LOAD SSSSH,SP
000405   336 010              LOAD IX,SSSSH
000407   010                  INC IX
000410   234 000              COMP IX,LH
000412   047 002              JUMP,EQ .+4
000414   062                  POP A
000415   063                  POP B
000416   336 004              LOAD IX,SSH
000420   337 000              LOAD LH,IX
000422   236 002              LOAD SP,SH
000424   064                  DEC SP
000425   067                  PUSH B
000426   066                  PUSH A
000427   336 006              LOAD IX,SSSH
000431   156 000              JUMP (IX)+0
                              INCAFT:
000433   227 003              LOAD SL,A
000435   336 002              LOAD IX,SH
000437   246 001              LOAD A,(IX)+1
000441   066                  PUSH A
000442   033                  ADD A,B
000443   247 001              LOAD (IX)+1,A
000445   346 000              LOAD B,(IX)+0
000447   067                  PUSH B
000450   311 000              ADDC B,#0
000452   347 000              LOAD (IX)+0,B
000454   062                  POP A
000455   063                  POP B
000456   071                  RET
                              .END


assembly complete              start adress  0000
      assembly time    0.024   seconds
```

C H A P I T R E IV

PROCEDURE D'IMPLEMENTATION POUR

UN PROCESSEUR ET PARAMETRISATION

Dans ce chapitre,nous présentons les procédures nécessaires à la paramétrisation du cross-compilateur et à sa mise en oeuvre pour un processeur.

Le premier paragraphe présente la procédure d'implémentation au niveau du cross-assemblage.Nous décrivons les différentes opérations à réaliser:définition du langage assembleur CALM du processeur que l'on veut utiliser et réalisation de la description UNIASS pour ce même processeur.

Le deuxième paragraphe décrit la procédure d'implémentation au niveau de la traduction CALM de VAC.Il convient tout d'abord de définir l'algorithme CALM qui correspond à chaque instruction VAC.Il faut ensuite choisir les procédures de traduction:utilisation de sous-routines ou non.On peut alors paramétrer la première passe de la phase de traduction. Pour paramétrer la seconde passe,il convient avant tout de déterminer les séquences CALM à supprimer ou réduire et de définir les séquences d'initialisation et de terminaison du programme traduit.

Dans le troisième paragraphe,nous abordons le problème de la programmation des programmes C.Celle-ci est en effet influencée par le processeur pour lequel les programmes sont écrits.Ce problème sera traité de façon très générale,car nous n'avons pu,faute de temps,l'approfondir dans le cadre de ce mémoire.

IV.I Procédure d'implémentation au niveau du cross-assemblage

Dans ce paragraphe,nous définissons les actions à réaliser pour paramétrer le cross-compilateur au niveau du cross-assemblage.Nous expliquons dans un premier point comment définir le langage assembleur CALM du processeur qu'on veut utiliser.Ensuite,nous présentons la manière de réaliser la description du jeu d'instructions pour UNIASS.

IV.II Définition du langage assembleur CALM

La première opération à réaliser est de définir le jeu d'instructions du processeur à l'aide de la syntaxe de CALM. Cette description peut être réalisée à partir du jeu d'instructions standard du processeur.Il s'agit en fait d'un simple changement de notations,fait en respectant la syntaxe de CALM.Il faut noter que l'utilisation de CALM n'est pas obligatoire pour utiliser UNIASS.C'est un choix que nous avons fait pour les raisons données dans le chapitre I.

Nous allons expliciter plus en détail les règles et conventions propres à l'assembleur CALM.Elles portent sur trois parties:les registres,les modes d'adressages et les mnémonics des instructions.

- les registres

On retrouve généralement:

. le program counter PC
. le stack pointer SP
. le registre d'état F
. les registres d'index
. les accumulateurs et autres registres

Il convient de donner un nom symbolique à chaque registre.CALM propose de donner un nom à une lettrepour un registre 8 bits,un nom à deux lettres pour un registre I6 bits. (voir table I)

- les modes d'adressage

Le mode d'adressage peut être direct.L'adresse est alors donnée dans le champs de l'instruction.Le mode d'adressage direct peut prendre trois formes:adressage relatif (le déplacement par rapport au program counter est donné dans le champs de l'instruction),adressage absolu dans la page 0 de la mémoire (adresse 8 bits donnée dans le champs de l'instruction) et adressage absolu étendu (adresse I6 bits donnée dans le champs de l'instruction).

Les deux autres modes d'adressage les plus importants sont l'adressage indexé et l'adressage indirect.Ces deux modes d'adressage sont fort semblables:dans l'adressage indexé,l'adresse se touve dans un registre;dans l'adressage indirect,elle se trouve dans une case mémoire.

Il existe parfois certaines variantes de ces modes d'adressage comme l'adressage indexé avec post ou pré-incrément ou post ou pré-décrément.

CALM propose toute une série de notations pour représenter ces différents modes d'adressage.Ces conventions sont reprises dans la table II.

- les mnémonics des instructions

On p ut grouper les instructions en cinq catégories: transferts de données,instructions à deux opérandes,instruc-

tions arythmétiques à I opérande,instructions de branchements et d'appel de sous-routines et instructions de contrôle du processeur.

. transferts de données

Trois types d'instructions sont repris dans cette catégorie:l'instruction LOAD qui permet de placer une valeur dans un registre ou à une adresse en mémoire à partir d'un registre ou d'une adresse mémoire;l'instruction EX qui permet d'échanger les valeurs contenues dans des registres ou en mémoire;les instructions PUSH et POP qui sont des instructions de transfert avec la pile.(voir table III)

. instructions à deux opérandes

On peut trouver une première sous-catégorie:les instructions ADD et SUB (addition et soustraction).Ces deux mnémonics peuvent être préfixés ou postfixés pour indiquer le type d'opération désiré.(exemple:addition avec carry, soustraction décimale).Les conventions pour préfixer et postfixer se trouvent dans la table I.

Dans les autres instructions à deux opérandes,on peut citer les opérations logiques,la multiplication,la division,la comparaison.(voir table III)

. instructions à une opérande

Cette catégorie reprend principalement les opérations de shift,d'incrément,de décrément,de remise à 0,de positionnement de bits.(voir table III et IV)

. instructions de branchements et d'appel de sous-routines

On retrouve ici les instructions de branchement conditionnel ou inconditionnel.La condition de branchement dépend de la valeur du registre d'état.La table I définit les mnémonics utilisés pour tester les différents cas possibles.

. instructions de contrôle du processeur

Ce sont les instructions d'interruption,d'arrêt,d'attente,d'exécution.(voir table IV)

Avec l'aide de ces règles et conventions,il convient de définir le jeu d'instructions CALM pour le processeur qu'on veut paramétrer.A titre d'exemple,on peut rappeler celui que nous avons donné pour l'Intel 8080 pour les instructions d'addition.

Les instructions standards étaient les suivantes:

ADD [A,B,C,D,E,H,L] (registres 8 bits) ADD (HL) (adresse dans HL) ADD m(adresse I6 bits) ADI d(valeur immédiate)	addition 8 bits dans l'accumulateur A
ADC [A,B,C,D,E,H,L] ADC (HL) ADC m ADC d	addition 8 bits avec carry dans l'accumulateur A
DAD [BC,DE,HL,SP] (registres I6 bits)	addition I6 bits dans le registre HL

En utilisant la syntaxe de CALM on obtient respectivement:

```
ADD A, [A,B,C,D,E,H,L]
ADD A,(HL)
ADD A,m
ADD A,#d
ADDC A, [A,B,C,D,E,H,L]
ADDC A,(HL)
ADDC A,m
ADDC A,#d
ADD HL, [BC,DE,HL,SP]
```

REGISTERS

Program counter	PC
Stack pointer (16 bits)	SP
Index register (16 bits)	XX,IX,IY
8-bit register	A,B,...,R0,R1,R2,...
16-bit register	HL,IX,...P0,P2,...
Flag register (8 bits)	F
Flag bits	carry C, link L, sign S, overflow V, zero value Z, interrupt I, decimal mode D

MEMORY REFERENCE

Location (or its contents) of address m	m
Loc. (or its contents) the address of which is the contents of X	(X)
Loc. (or its contents) the address of which is in the location of address m	@m

PREFIXES TO INSTRUCTIONS

Inverted operands	I
Decimal operation	D
Floating point operation	F

POSTFIXES TO INSTRUCTIONS

With carry	C
With link bit	L
Byte operation (in a 16-bit µP)	B
Followed by a jump if test t is true	,Jt
Followed by a skip if test t is true	,St

TESTS t IN CONDITIONAL JUMP OR SKIP INSTRUCTIONS

If carry set	C=1	,CS	or	lower	,LO
If carry clear	C=0	,CC		higher or same	,HS
If sign bit set	S=1	,SS	or	minus result	,MI
If sign bit clear	S=0	,SC		positive result	,PL
If zero bit set	Z=1	,ZS	or	result equal to zero	,EQ
If zero bit clear	Z=0	,ZC		result non equal to zero	,NE
If overflow set	V=1	,VS			
If overflow clear	V=0	,VC			

If higher	CvZ=0	(>)	,HI
If higher or equal	C=0	(≥)	,CC or ,HS
If equal	Z=0	(=)	,EQ
If lower or same	CvZ=1	(≤)	,LS
If lower	C=1	(<)	,CS or ,LO

For two positive numbers after a COMP or SUB instr.

If greater	Zv(S⊕V)=0	(>)	,GT
If greater or equal	S⊕V=0	(≥)	,GE
If equal	Z=0	(=)	,EQ
If lower or equal	Zv(S⊕V)=1	(≤)	,LE
If lower than	S⊕V=1	(<)	,LT

For two signed numbers after a COMP or SUB instr.

SIGNS

add	+	NOT	¬
sub	-	AND	∧
multiply	*	OR	v
divide	/	Exclusive OR	⊕

NUMERIC EXPRESSIONS

4-bit expression	i
8-bit interger expression	n
8-bit signed expression	n'
16-bit integer expression	m
15-bit signed expression	m'

Table I. Assembly language conventions.

ADDRESSING MODE	SYMBOLIC ASSEMBLER NOTATION	ADDRESSING OPERATION	8080 Intel, AMD, TI	Z-80 Zilog	6800 Motorola, AMI	650x MOS	2650 Signetics	1801 RCA	1802 RCA	F-8 Fairchild, Mostek	SC/MP NS	9002 FA		1000 TI		PACE NS	9900 TI	LSI 11 DEC
IMMEDIATE	# VALUE	#i (4 bits)						●	●	●				●				
		#n (8 bits)	●	●	●	●	●	●	●	●	●	●					●	●
		#m (16 bits)	●	●	●					●						●	●	●
(Implied)	R0,A,XX (pre-defined symbol)	R0,A,XX	●	●	●	●	●	●	●	●	●	●		●		●	●	●
ABSOLUTE	ADDRESS	m (14-16 bits, complete address)	●	●	●	●	●		●	●	●	●					●	●
		n (8 bits, page 0)			●	●				●						●		
		:n (8 bits, same page)						●	●					●				
RELATIVE		(PC)+n' or .+n' (8-bit displ.)		●	●	●	●			●						●	●	
		(PC):+n or .:+n ($^{8\text{-bit, wrap around,}}_{\text{in same page}}$)									●							
		(PC)+m' or .+m' (16-bit displ.)																●
(Register)		(XX) (no displacement)	●	●				●	●	●		●						●
INDEXED	(XX) + DISPL	(XX)+n (8-bit positive displ.)			●													
		(XX)+n' (8-bit displacement)		●												●		
	(XX+) post incremented (XX-) post decremented (+XX) pre incremented (-XX) pre decremented	(XX)+m' (16-bit displacement)															●	●
		(X)+n' (8-bit displ.)																
		(X):+n ($^{8\text{-bit, wrap around,}}_{\text{in same page}}$)				●					●			●				
		(X)+m' (16-bit displ.)				●	●											
INDIRECT	@ ADDRESS	@m					●									●		
		@(PC)+n' or @.+n'					●											
		@(PC)+m' or @.+m'																●
INDEXED INDIRECT (PRE-INDEXED)	@(XX) + DISPL	@(XX)+n'															●	
		@(X)+m				●	●											
		@(XX)+m'																●
INDIRECT INDEXED (POST-INDEXED)	(XX) + @ADDISP	(X)+ @n				●												
		(XX)+ @m																

Table II. Assembly language addressing mode conventions.

			8080 Intel, AMD, NS	Z-80 Zilog	6800 Motorola, AMI	650x MOS	2650 Signetics	1801 RCA	1802 RCA	F 8 Fair., Mostek	SC/MP NS	9002 EA		1000 TI		PACE NS	9900 TI	LSI 11 DEC
Load d with s	d ← s	LOAD d,s	●	●	●	●	●	●	●	●	●	●		●		●	●	●
Exchange d and s	d ↔ s	EX d,s	●	●						●		●		●		●		
Save d on stack (LOAD (SP-),d)		PUSH d	●	●	●	●										●		●
Load d with top of stack (LOAD d,(SP+))		POP d	●	●	●	●										●		●
Add d with s, result in d	d ← d+s	ADD d,s	●	●	●		●	●	●	●				●		●	●	●
Add with carry	d ← d+s+c	ADDC d,s	●	●	●	●	●		●	●	●	●				●		
Decimal add	d ← d+s	DADD d,s								●								
Decimal add with carry	d ← d+s+c	DADDC d,s									●					●		
Subtract	d ← d-s	SUB d,s	●	●	●		●	●	●								●	●
Subtract with carry	d ← d-s-c	SUBC d,s	●	●	●	●	●		●							●		
Inverted subtract	d ← s-d	ISUB d,s						●	●					●				
Add 1's complement	d ← d+¬s	ACO d,s																
Add 1's complement with carry		ACOC d,s									●							
Inverted add 1's compl.	d ← s+¬d	IACO d,s														●		
Multiply	d ← d·s	MUL d,s															●	
Divide	d ← d : s	DIV d,s															●	
Logical AND	d ← d∧s	AND d,s	●	●	●	●	●	●	●	●	●	●				●		●
Bit clear	d ← d∧¬s	BIC d,s														●	●	●
Logical OR (bit set)	d ← d∨s	OR d,s	●	●	●	●	●	●	●	●	●	●				●	●	●
Exclusive OR	d ← d⊕s	XOR d,s	●	●	●	●	●	●	●	●	●	●					●	●
Exclusive NOR	d ← d⊕¬s	XNOR d,s																
Compare (subtract - no load)	d-s	COMP d,s	●	●	●	●	●			●		●		●		●	●	●
Bit test (and - no load)	d∧s	BIT d,s			●	●	●									●	●	●
Rotate right d		RR d	●	●			●				●	●				●	●	
Rotate left d		RL d	●	●			●					●				●	●	
Rotate right through carry		RRC d	●	●	●	●	●		●		●	●				●		●
Rotate left through carry		RLC d	●	●	●	●	●		●			●				●		●
Shift right		SR d								●	●					●		
Shift left		SL d								●						●	●	
Shift right trough carry		SRC d			●	●		●	●							●		
Shift left through carry		SLC d			●	●			●							●		●
Arithmetic shift right		ASR d			●												●	●
Arith. shift right from carry		ASRC d									●							
Swap (half-)bytes		SWAP d															●	●

Table III. Microprocessor instructions list (I).

Table IV. Microprocessor instruction list (II).

		3030 Intel, AMD, TI	Z 80 Zilog	6800 Motorola, AMI	650X MOS	2650 Signetics	1801 RCA	1802 RCA	F 8 Fair., Mostek	SC/MP NS	9002 FA		1000 TI		PACE NS	9900 TI	LSI 11 DEC
Clear d	d ← 0	CLR d		●					●		●					●	●
Clear bit i of d	$d_i \leftarrow 0$	CLR d:i		●			●						●			●	
Clear flag bit b	b ← 0	CLRb			●	●		●			●	●				●	
Set bit i of d	$d_i \leftarrow 1$	SET d:i					●						●			●	
Set flag bit b	b ← 1	SETb	●	●	●	●		●			●	●			●	●	
Pulse b	b ← 0 ← 1	PULb									●				●		
Complement d (1's compl.)	d ← −d	CPL d	●	●	●					●		●				●	●
Negate d (2's compl.)	d ← -d	NEG d			●								●			●	●
Absolute value of d	d ← |d|	ABS d														●	
Decimal adjust after add.		DAA d	●		●												
Decimal adjust after sub.		DAS d				●											
Decimal adjust (add or sub)		DA d		●													
Increment d	d ← d+1	INC d	●	●	●	●		●	●	●		●		●		●	●
Increment d and jump to a if t true		INCJ,t d,a					●					●			●		
Increment d and skip if t true		INCS,t d															
Decrement d		DEC d	●	●	●	●		●	●	●		●		●		●	●
Decrement d and jump to a if t true		DECJ,t d,a		●			●					●			●		●
Decrement d and skip if t true		DECS,t d															
Test d (load flags)	d ← d	TEST d			●								●				●
Test bit i of d (load Z flag)		TEST d:i		●			●						●				
Jump (inconditional) to a		JUMP a	●	●	●	●	●	●	●		●	●		●		●	●
Jump to a if test t true		JUMP,t a	●	●	●	●	●	●	●		●	●		●		●	●
Skip next instruction		SKIP						●	●								
Skip next instr. if t true		SKIP,t															
Long skip of two memory words		LSKIP							●								
Call subroutine at a		CALL a	●	●	●	●	●			●		●		●		●	●
Call routine at a if t true		CALL,t a	●	●			●							●			
Return from subroutine		RET	●	●	●	●	●			●		●		●		●	●
Return if test t true		RET,t	●	●			●										
Trap (at interrupt vector v)		TRAP (v)			●	●											●
Return from interrupt		RTI		●	●	●		●	●						●		●
Execute single instr. at a		EXEC a														●	
Interrupt ON (clear int. mask)		ION	●	●	●	●				●	●	●					●
Interrupt OFF (set int. mask)		IOF	●	●	●	●				●	●	●					●
No operation		NOP	●	●	●	●	●		●	●	●	●				●	●
Delay proportional to d value		DELAY d								●	●				●		
Wait for an interrupt		WAIT	●	●	●		●	●	●							●	●
Halt (wait for an ext. interrupt)		HALT													●		●
Reset external devices		RESET						●	●							●	●
Number of add. mnemonics required		0	14	0	0	0	2	3	0	2	1	0	1	0	4	8	8

IV.I2 Réalisation de le description UNIASS

Après avoir défini le jeu d'instructions CALM pour le processeur à paramétrer,il convient de réaliser la description de ce jeu d'instructions pour UNIASS à l'aide des pseudo-instructions que nous avons décrites en III.3I2: .MENU, .CODE, .TEST, .MAXI, .INSTR.

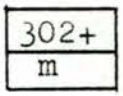
Pour mettre au point cette description,il convient de respecter un certain nombre de règles et de suivre certains conseils.

- la définition des pseudo-instructions .MENU, .CODE, .TEST, .MAXI ne doivent pas nécessairement apparaître dans le texte avant leur utilisation dans une pseudo-instruction .INSTR.

- il est recommandé de placer les chaînes de caractères avant les menus quand ceux-ci peuvent apparaître à la même place dans une instruction.L'assemblage sera plus rapide car la reconnaissance d'une chaîne de caractère réussit ou rate immédiatement;mais la reconnaissance d'une chaîne de caractères dans un menu est plus lente:l'assembleur essaie de reconnaître chaque chaîne de caractères du menu et ce de façon séquentielle.

exemple:instruction JUMP de l'Intel 8080

351	JUMP (HL)
303	JUMP m
m	



JUMP,t m

0	ZC NE
I0	ZS EQ
20	HQ CC
30	LO CS
60	PL SC
70	MI SS
40	VC PO
50	VS PE

```
.MENU TSTIZ = ('NE'=0,'ZC'=0,'EQ'=I0,'ZS'=I0,'HQ'=20,
'CC'=20,'LO'=30,'CS'=30,'PL'=60,'SC'=60,'MI'=70,'SS'=70;
'VC'=40,'PO'=40,'VS'=50,'PE'=50)
.INSTR 'JUMP'_'(HL)'BI(35I)
.INSTR 'JUMP'_(TSTIZ)BI(V)_E(I6.)$W2BI(302)
.INSTR 'JUMP'_E(I6.)$W2 BI(303)
```

N.B.: W2 est un .MAXI que nous avons décrit en III.3I2,il transforme un nombre I6 bits en deux bytes B2 et B3

En plaçant la reconnaissance de JUMP (HL) avant la reconnaissance des instructions de branchement conditionnel l'assemblage est accéléré.Dans le cas contraire,l'assembleur devrait parcourir toutes les chaînes de caractères du menu TSTIZ avant de reconnaître la chaîne '(HL)' chaque fois qu'une instruction JUMP (HL) est rencontrée.En effet l'assembleur travaille séquentiellement.Dans le cas d'une instruction de branchement conditionnel,le travail supplémentaire est faible puisque l'assembleur testera uniquement la chaîne '(HL)' en trop.

- les chaînes de caractères les plus longues doivent venir avant les plus courtes si leurs premiers caractères sont identiques.Cette règle est valable pour les chaînes

de caractères dans les menus ou pour des chaînes se trouvant directement dans les instructions .INSTR.

exemple:le cross-assembleur doit d'abord pouvoir reconnaître le mnémonic ADDC avant ADD sinon ADD sera toujours reconnu et le reste de la chaîne conduira à une erreur.

- quand on décrit différents types de paramètres qui peuvent apparaître à la même place dans une instruction,les expressions doivent toujours venir à la fin.Sinon quel que soit le paramètre qui apparaît,l'assembleur considère que c'est une expression.

exemple: .INSTR 'JUMP'_(TSTIZ)BI(V)_E(I6.)$W2 BI(302)
.INSTR 'JUMP'_E(I6.)$W2 BI(303)

Si les instructions sont inversées,le jump conditionnel ne sera jamais testé;l'assembleur cherche une expression I6 bits.

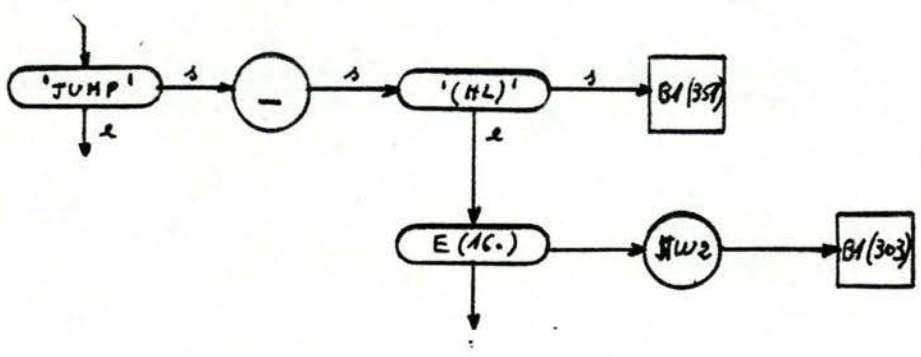
- si différents types d'expressions peuvent apparaître à la même place,les expressions les plus courtes doivent venir en tête.

- pour des raisons d'optimisation,il est intéressant de placer en tête dans les menus les chaînes de caractères les plus souvent utilisées.De même,les instructions les plus utilisées seront placées en tête de la description.L'assembleur travaille en effet de façon séquentielle.Pour bien comprendre ceci,montrons comment l'assembleur construit l'arbre à partir de la description.

Soit les instructions:

.INSTR 'JUMP'_'(HL)'BI(351)
.INSTR 'JUMP'_E(I6.)$W2 BI(303)

L'arbre correspondant est le suivant:



On voit que la première instruction comprenant (HL) est d'abord testée.Si elle apparaît souvent,il est donc intéressant de la décrire avant.

IV.2 Procédure d'implémentation au niveau de la traduction CALM de VAC

Dans ce paragraphe,nous présentons les opérations à réaliser pour paramétrer le cross-compilateur au niveau de la traduction en code CALM des instructions VAC.Nous donnons également quelques conseils susceptibles d'améliorer les performances du code CALM généré.

Il convient d'abord de définir les algorithmes CALM pour chaque instruction VAC.Ensuite,il faut déterminer pour chaque algorithme s'il est possible et intéressant d'en faire une sous-routine ou s'il faut au contraire générer tout le code chaque fois que l'instruction VAC est rencontrée dans le programme à traduire.Il faut également déterminer les optimisations réalisables.Après toutes ces opérations,on peut paramétrer les deux programmes Snobol de traduction et d'optimisation.

IV.2I Définition des algorithmes CALM

Comme nous l'avons vu dans le chapitre précédant,il convient de construire l'algorithme CALM qui correspond à chaque instruction VAC.Pour cela,on dispose du jeu d'instructions CALM pour le processeur qu'on veut utiliser,jeu d'instructions qui a été défini lors de la définition du langage assembleur CALM du processeur dans le paragraphe précédant.

Pour écrire ces algorithmes,il convient donc de très bien connaître ce jeu d'instructions et par conséquent les possibilités du processeur.il est également important de très bien connaître les modes d'adressages disponibles et les effets de chaque instruction sur le registre d'état du processeur.

Quand à la manière d'écrire les algorithmes,il est important d'essayer d'être le plus systématique possible;c'est-à-dire d'utiliser le plus possible les mêmes instructions, les mêmes zones mémoires ou registres,pour réaliser des fonctions identiques dans des algorithmes correspondants à des instructions VAC différentes - par exemple toujours utiliser le ou les mêmes registres pour sauver ou extraire des valeurs de la pile.Ceci permettra de réaliser certaines optimisations sur lesquelles nous reviendrons plus loin.

Donnons,à titre d'exemple,un algorithme possible pour les instructions VAC ,PLUSI et EQI, pour l'Intel 8080.Nous avons déjà fourni ces deux algorithmes pour le Motorola 6800.

- instruction PLUSI

```
POP HL        ;extrais l'opérande 2 dans HL
POP BC        ;extrais l'opérande I dans BC
ADD HL,BC     ;(HL)+(BC) ->HL
PUSH HL       ;sauve le résultat sur la pile
```

- instruction EQI

```
POP HL        ;extrais l'opérande 2 dans HL
POP BC        ;extrais l'opérande I dans BC
LOAD A,B      ;place le byte le plus significatif
               de l'opérande I dans A
COMP A,H      ;compare les parties les plus si-
               gnificatives des 2 opérandes
JUMP,ZC .+9   ;branche si résultat faux
LOAD A,L      ;place le byte le moins significatif
               de l'opérande I dans A
LOAD HL,#I    ;I -> HL
COMP A,C      ;compare les parties les moins si-
               gnificatives des 2 opérandes
JUMP,ZS .+5   ;branche si résultat vrai
```

```
LOAD HL,#0    ;0 -> HL
PUSH HL       ;sauve le résultat sur la pile
```

IV.22 Choix des procédures de traduction

Afin d'obtenir un texte CALM suffisamment performant, il faut déterminer pour chaque algorithme CALM s'il est possible et intéressant d'utiliser une sous-routine qui ne sera générée qu'une seule fois par programme traduit - si l'instruction VAC qui lui correspond se trouve dans le programme - et par conséquent de ne générer que les instructions strictement nécessaires ou qui ne peuvent faire partie de la sous-routine chaque fois que l'instruction VAC apparaît dans le texte du programme à traduire.

Des règles précises sont difficilement applicables pour savoir s'il faut ou non avoir recours à une sous-routine. On peut cependant rappeler que les instructions de manipulation de la pile ou de modification du stack pointer doivent être utilisées avec précaution à l'intérieur d'une sous-routine car les instructions d'appel et de retour d'une sous-routine modifient l'état de la pile et du stack pointer.Les extractions et sauvetages doivent en général être utilisés en dehors de telles sous-routines sous peine de compliquer grandement la gestion de la pile et donc de devenir moins performant.

A titre d'exemple,on peut choisir pour l'instruction PLUSI de générer les 4 instructions CALM décrites en IV.2I chaque fois que PLUSI est rencontré dans le programme à traduire.L'utilisation d'une sous-routine n'apporterait rien dans ce cas.En effet, une seule instruction ne manipule pas la pile.D'autre part le nombre d'instructions à générer est très faible.

Par contre,on peut décider pour l'instruction EQI d' utiliser une sous-routine.Le texte généré à chaque rencontre de l'instruction VAC dans le texte à traduire devient:

```
POP HL
POP BC
CALL EQI
PUSH HL
```

La sous-routine est la suivante:

```
EQI:
LOAD A,B
COMP A,H
JUMP,ZC .+9
LOAD A,L
LOAD HL,#I
COMP A,C
JUMP,ZS .+5
LOAD HL,#0
RET
```

On remarque que le gain peut être appréciable si le nombre d'instructions EQI du programme VAC est important. On génère seulement 4 instructions chaque fois au lieu de II si l'on n'utilise pas de sous-routine.La perte en temps d'exécution,représentée par l'exécution des instructions CALL et RET est le prix qu'il faut payer.

IV.23 Paramétrisation de la phase de traduction CALM

Après avoir défini tous les algorithmes CALM et après avoir choisi les séquences d'instructions CALM à générer pour chaque instruction VAC,il est possible de paramétrer le programme qui assure la traduction en CALM.Rappelons que

ce programme a été décrit en III.222 et qu'il réalise les fonctions suivantes:

- lecture ligne par ligne du fichier VAC
- reconnaissance de la ligne lue
- en fonction de cette reconnaissance générer l'algorithme CALM qui correspond sur le fichier output

Il n'y a que cette troisième partie qui doit être modifiée.Nous allons le montrer sur un exemple.Nous avons donné en III.222 un exemple pour le Motorola 6800 (pour les instructions PLUSI et EQI).Nous allons présenter ce même programme pour l'Intel 8080.

```
goI     input = syspit                  /f(end)
        input */8* 'PLUSI'              / s(plusi)
        input */8* 'EQI'                /s(eqi)f(goI)

plusi   syspot = 'POP HL'
        syspot = 'POP BC'
        syspot = 'ADD HL,BC'
        syspot = 'PUSH HL'              /(goI)

eqi     syspot = 'POP HL'
        syspot = 'POP BC'
        syspot = 'CALL EQI'
        syspot = 'PUSH HL'              /(goI)

end     syspot = ''
```

On remarque que la structure du programme comparée à celle donnée pour le Motorola 6800 est la même.Les seules modifications concernent les séquences d'instructions CALM à générer (ce qui est encadré dans notre exemple).

IV.24 Optimisations et paramétrisation de la phase d'optimisation

Pour paramétrer la seconde passe de la phase de traduction,nous avons déjà déterminé les sous-routines que nous voulons utiliser.Il reste encore à déterminer les séquences générées par la première passe qui peuvent éventuellement être supprimées ou réduites et à écrire les séquences d'initialisation et de terminaison de programme.

IV.24I Détection des séquences à supprimer ou à réduire

La façon la plus simple de détecter ces séquences est de réaliser un certain nombre de tests,c'est-à-dire d'exécuter la première passe de la phase de traduction pour un certain nombre de programmes et d'analyser le texte généré afin de découvrir ces séquences.

On peut également observer les instructions de début et de fin des algorithmes CALM.C'est en effet lors de l'enchaînement de plusieurs instructions VAC que des optimisations sont possibles.

Pour l'Intel 8080,nous avons découvert deux séquences d'instructions CALM qui peuvent être supprimées:

PUSH HL	PUSH BC
POP HL	POP BC

IV.242 Séquences d'initialisation et de terminaison

Rappelons que les opérations essentielles de début de programme consistent à réserver les variables éventuelles utilisées,à initialiser l'adresse de base locale et le stack pointer et de brancher à l'étiquette MAIN,véritable début

du programme.En fin de programme,il faut arrêter l'exécution du programme.

exemple pour l'Intel 8080

```
        .RDX IO.        ;toutes les valeurs en base IO
        .LOC DEBUT      ;l'exécution débute ici
DEBUT:  LOAD HL,ADR     ;initialisation de l'adresse de base
                         locale
        LOAD SP,HL      ;initialisation du stack pointer
        JUMP MAIN
FIN:    WAIT            ;arrêt de la machine
```

IV.243 Paramétrisation de la phase d'optimisation

Nous disposons maintenant de tous les éléments nécessaires pour paramétrer la phase d'optimisation.Nous connaissons les sous-routines,les séquences à supprimer et la séquence d'initialisation et de terminaison.

- initialisation des variables globales

Rappelons qu'il faut une variable globale pour chaque sous-routine afin de déterminer si elle doit être générée. On peut conseiller de choisir un nom explicite pour chaque variable,par exemple le nom de la sous-routine.Ces variables sont initialisées à 0.

exemple: eqi ='0'

- lecture du fichier input

Le nombre d'instructions à lire "ensemble" dépend du nombre d'instructions des séquences à supprimer ou à modi-

fier.Pour l'Intel 8080,il faut lire 2 lignes à la fois.Nous obtenons donc:

```
readI       inputI = syspit      /f(finI);branche si fin de
                                          fichier
read2       input2 = syspit      /f(fin2);branche si fin de
                                          fichier
```

- recherche des séquences à supprimer ou à modifier

Il convient de tester si le buffer d'instructions lues comprend une séquence à supprimer ou modifier.Pour l'Intel 8080 nous obtenons:

```
testI       inputI  ** 'PUSH HL'     /f(test2)
            input2  ** 'POP HL'      /s(readI)
test2       inputI  ** 'PUSH BC'     /f(chang)
            input2  ** 'POP BC'      /s(readI)

chang       ...
            syspot = inputI
            inputI = input2          /(read2)
```

En 'chang' ,la première ligne lue (dans inputI) peut être recopiée sur le fichier output.Il faut alors ajuster le buffer.

- déterminer si une instruction contient un appel de sous-routine

Rappelons que ceci est réalisé par la fonction appel. fnc(input) qu'il convient d'appeler avant d'écrire une ligne dans le fichier output.

Le programme précédant devient donc:

```
chang       appel.fnc(inputI)
            syspot = inputI
            inputI = input2              /(read2)
```

Pour écrire la fonction appel.fnc(input) ,on peut se référer à la description donnée pour le Motorola 6800.Elle sera construite de la même façon.

- <u>génération des sous-routines</u>

Cette fonction est réalisée par test.fnc qui est appelée lorsque tout le fichier input a été lu.Sa structure est la même que celle décrite pour le Motorola 6800 et ne demande pas plus de commentaires.

IV.3 Techniques de programmation C

Un point important pour l'optimisation des programmes, tant du point de vue de la place mémoire que du temps d'exécution est le choix des techniques de programmation des programmes C adaptées au processeur que l'on veut utiliser.

Nous ne donnons ici qu'une introduction à ce problème. Il n'a pas pu,faute de temps,être approfondi dans le cadre de ce mémoire.Il est cependant certain que ce problème est essentiel si l'on désire obtenir des programmes performants.

Il convient,une fois que le cross-compilateur est correctement paramétré et testé,de déterminer les techniques de programmation les plus adéquates:faut-il utiliser les instructions spéciales d'incrément ou de décrément,les boucles for sont elles plus performantes que les instructions while,les instructions switch sont elles intéressantes et dans quelle mesure;il faut donc se poser un certain nombre de questions.Pour y répondre,la meilleure solution est sans doute de les tester et de comparer les résultats des programmes objets obtenus après cross-compilation,tant du point de vue de la taille des programmes que des temps d'exécution.

Il faut donc mettre au point un certain nombre de conseils de programmation liés au processeur que l'on utilise,en veuillant cependant à conserver des règles permettant une programmation claire et lisible.

A titre d'exemple,comparons les 2 instructions i++ et i=i+I qui peuvent souvent être utilisées indifféremment. Nous donnons respectivement le code généré pour i++ et i=i+I.

nstruction i++

```
000000   000                    LH:  .BYTE 0
000001   000                    LL:  .BYTE 0
000002   000                    SH:  .BYTE 0
000004   000                    SL:  .BYTE 0
000063   226 001                LOAD A,LL
000065   326 000                LOAD B,LH
000067   316 000 000            LOAD IX,#2*0
000072   337 002                LOAD SH,IX
000074   233 003                ADD A,SL
000076   331 002                ADDC B,SH
000100   327 002                LOAD SH,B
000102   306 001                LOAD B,#1
000104   275 000 211            CALL INCAFT
                                INCAFT:
000211   227 003                LOAD SL,A
000213   336 002                LOAD IX,SH
000215   246 001                LOAD A,(IX)+1
000217   066                    PUSH A
000220   033                    ADD A,B
000221   247 001                LOAD (IX)+1,A
000223   346 000                LOAD B,(IX)+0
000225   067                    PUSH B
000226   311 000                ADDC B,#0
000230   347 000                LOAD (IX)+0,B
000232   062                    POP A
000233   063                    POP B
000234   071                    RET
```

instruction i=i+1

```
000000    000                   LH:  .BYTE 0
000001    000                   LL:  .BYTE 0
000002    000                   SH:  .BYTE 0
000003    000                   SL:  .BYTE 0
000063    226 001               LOAD A,LL
000065    326 000               LOAD B,LH
000067    316 000 000           LOAD IX,#2*0
000072    337 002               LOAD SH,IX
000074    233 003               ADD A,SL
000076    331 002               ADDC B,SH
000100    066                   PUSH A
000101    067                   PUSH B
000102    336 000               LOAD IX,LH
000104    346 000               LOAD B,(IX)+2*0
000106    246 001               LOAD A,(IX)+1+2*0
000110    066                   PUSH A
000111    067                   PUSH B
000112    316 000 001           LOAD IX,#1
000115    337 002               LOAD SH,IX
000117    226 003               LOAD A,SL
000121    326 002               LOAD B,SH
000123    060                   INC IX,SP
000124    253 001               ADD A,(IX)+1
000126    351 000               ADDC B,(IX)+0
000130    061                   INC SP
000131    061                   INC SP
000132    060                   INC IX,SP
000133    061                   INC SP
000134    061                   INC SP
000135    356 000               LOAD IX,(IX)+0
000137    247 001               LOAD (IX)+1,A
000141    347 000               LOAD (IX)+0,B
```

On voit que i++ est de loin l'instruction la plus intéressante:elle ne comporte que 23 instructions assembleurs pour 28 à i=i+I.De plus,elle occupe 40 bytes pour 48 à i=i+I.Il y a donc un gain de place et un gain de temps d'exécution pour i++.

Nous croyons qu'il faut aussi se rendre compte que l'utilisation d'un tel cross-compilateur entraîne une diminution importante des performances si on les compare à celles d'un programme écrit directement en langage assembleur.

Voyons par exemple quelle est la traduction manuelle de l'instruction i=i+I (ou i++).Nous supposons que i se trouve dans les bytes SH et SL et est initialisée à 0.

```
000 000      000    SH: .BYTE 0
000 00I      000    SL: .BYTE 0
000 002 226 00I     LOAD A,SL
000 004 326 000     LOAD B,SH   ;charge SL et SH dans A et B
000 006 2I3 00I     ADD A, I
000 0I0 3II 000     ADDC B, 0   ;ajoute I à i
000 0I2 327 000     LOAD SH,B
000 0I4 227 00I     LOAD SL,A   ;place le résultat dans SH-SL
```

On voit qu'ici il faut seulement 6 instructions et I2 bytes.

C O N C L U S I O N

Pour tracer le bilan de ce travail,on peut dire que le cross-compilateur a été implémenté pour deux micro-processeurs:le Motorola 6800 et l'Intel 8080.

Pour réaliser ce cross-compliteur nous avons du:

- modifier le pré-compilateur VAC afin d'en améliorer les performances pour notre travail.

- écrire deux programmes en Snobol 3 qui assurent la traduction en code CALM et l'optimisation de ce même code CALM et réaliser leur paramétrisation pour les deux micro-processeurs cités ci-dessus.

- tester le programme de cross-assemblage UNIASS pour ces deux microprocesseurs.

D'autre part,le cross-compilateur a été testé pour le microprocesseur Motorola 6800.Un certain nombre de programmes cross-compilés ont été exécutés sur ce microprocesseur. Aucun test de ce genre n'a été réalisé pour l'Intel 8080.

Nous aimerions pour terminer,présenter et suggérer certaines améliorations qui pourraient être apportées au cross-compilateur existant.

Au niveau d'UNIASS,il serait sans aucun doute très intéressant d'implémenter le mode compilation qui permet de construire et stocker l'arbre utilisé par l'assembleur une fois pour toute.Cela permettrait un gain important de temps lors de la phase de cross-assemblage.

Il serait également intéressant que UNIASS accepte les caractères minuscules,ce qui n'est pas le cas actuellement.

Au niveau de la traduction en code CALM,l'écriture d'un macro-générateur permettrait également un gain de temps important,car comme nous l'avons déjà souligné,Snobol 3 est un interpréteur et il est très lent.

Enfin,au niveau de la programmation en C,il serait intéressant de mener une étude approfondie sur les performances des différentes instructions en fonction du microprocesseur.

D'autres problèmes pourraient également être envisagés:la possibilité de relier des programmes exécutables après leur passage dans UNIASS ou la création d'une librairie de routines.

B I B L I O G R A P H I E

- Uniass universal cross-assembler for microprocessors
  reference manual
- A universal cross-assembler for microprocessors
  A.M. Schmit,P. Schmit,M. Berthoud
  Swiss Federal Institute of Technology Lausanne I977
- Software facilities for microprocessors development and teaching
  J.D. Nicoud,A. Droz,R. Forster,D. Roux
  Swiss Federal Institute of Technology Lausanne I979
- A Common microprocessor assembly langage
  J.D. Nicoud
  Swiss Federal Institute of Technology Lausanne I976
- A proposed abstract machine for C
- Unix for beginners
  B.W. Kernighan
  Bell Telephone Laboratories
- Manuel utilisateur Unix
- Bell System Technical Journal
  Juillet-Août I978 Vol.57 n°6 part 2
  Bell Telephone Laboratories
- Programming in C - A tutorial
  B.W. Kernighan
  Bell Telephone Laboratories
- The C programming language
  B.W. Kernighan et D. Ritchie
  Bell Telephone Laboratories I978
- Microprocesseur Motorola 6800
  manuel de programmation
- The Snobol 3 programming language
  D.J. Farber, R.E. Griswold, I.P. Polonsky I966
- The M6 Macro-processor
  A.D. Hall
  Bell Telephone Laboratories

ANNEXE 1

JEU D'INSTRUCTIONS VAC

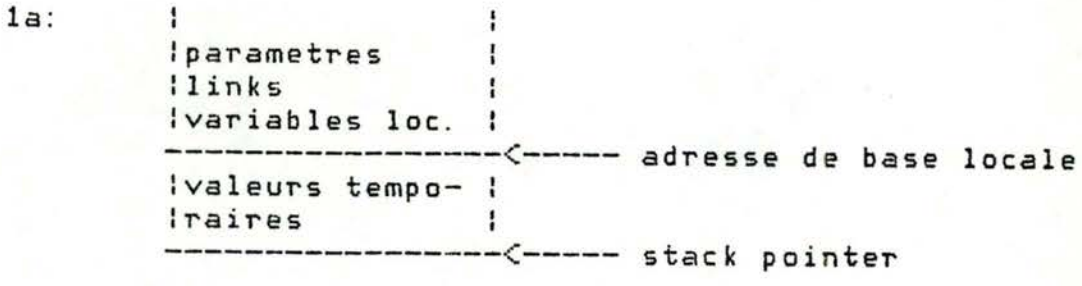
ANNEXE 1

JEU D'INSTRUCTIONS VAC

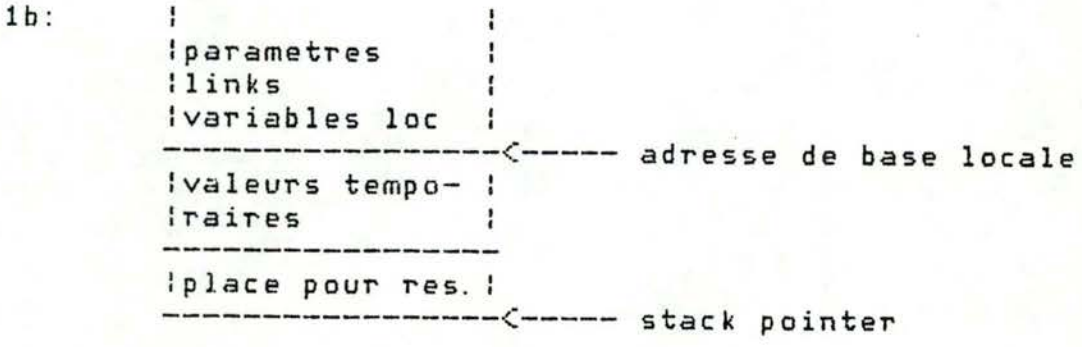
I Instructions de modification de l'environnement

operation	operande	operande sur la pile	pile av	pile ap	fonction
RRESI			1a	1b	reserver une place sur la pile pour placer le resultat d'une fonction
CALL	deplacem. de la fonction		1c	1d	placer l'adresse de retour sur la pile
CALLII		deplacement de la fonction	1c	1d	idem CALL
ENTRY	nom de la fonction		1d	1e	sauver l'adresse de base locale de la fonction appelante sur la pile
SAVE	nombre de bytes		1e	1f	calcul de la nouvelle adresse de base locale et du SP
RETRNI	nombre de bytes		2a ou 2b	2c	placer le resultat de la fonction a l'endroit reserve et restituer l'ancienne adresse de base locale

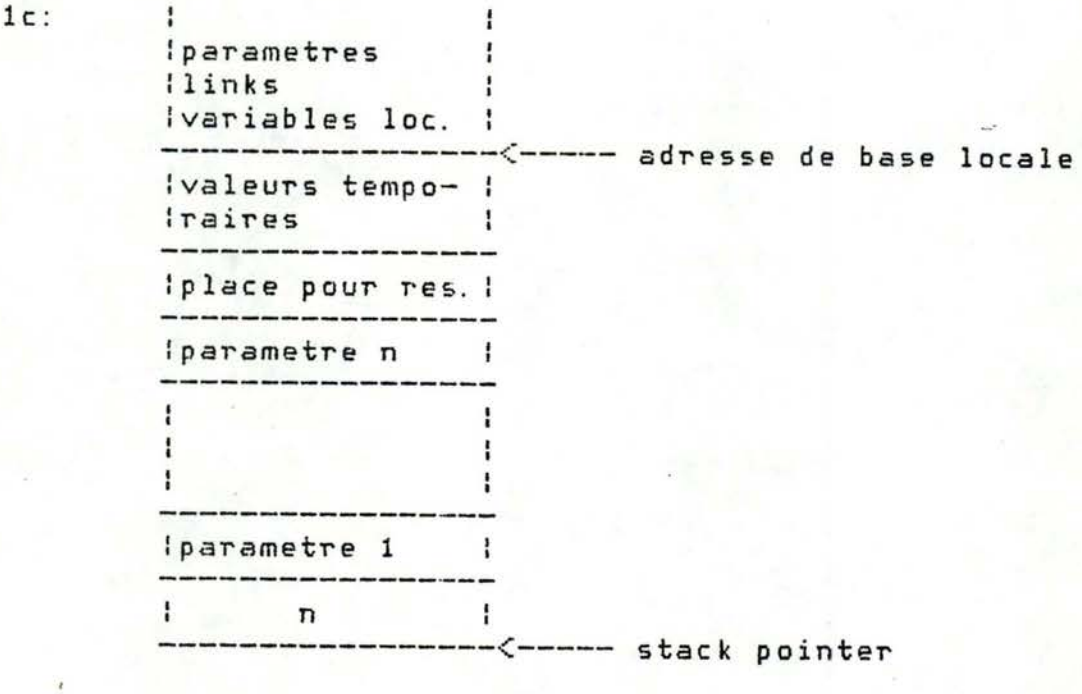
1a:

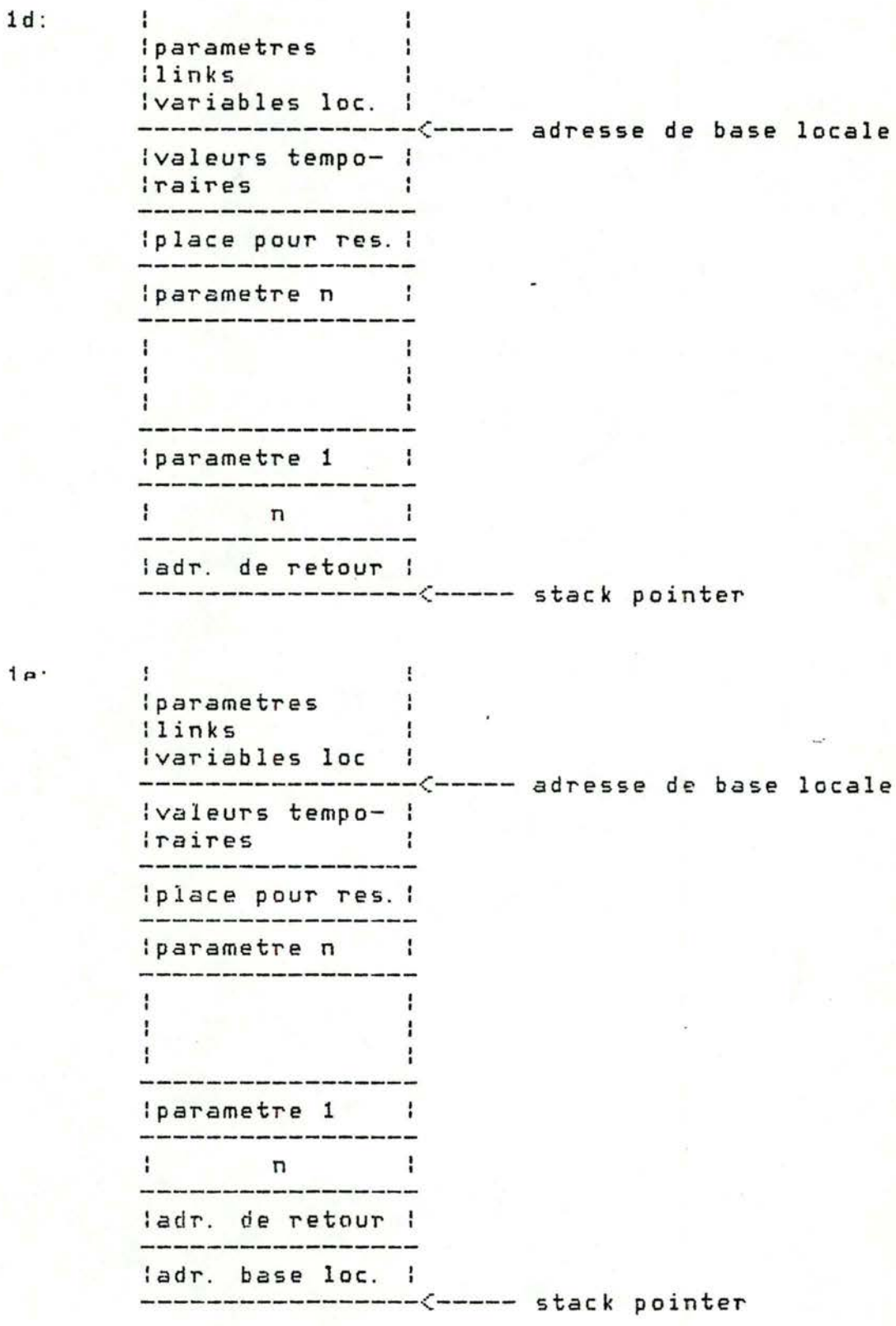

parametres
links
variables loc.
adresse de base locale
valeurs tempo-
raires
stack pointer

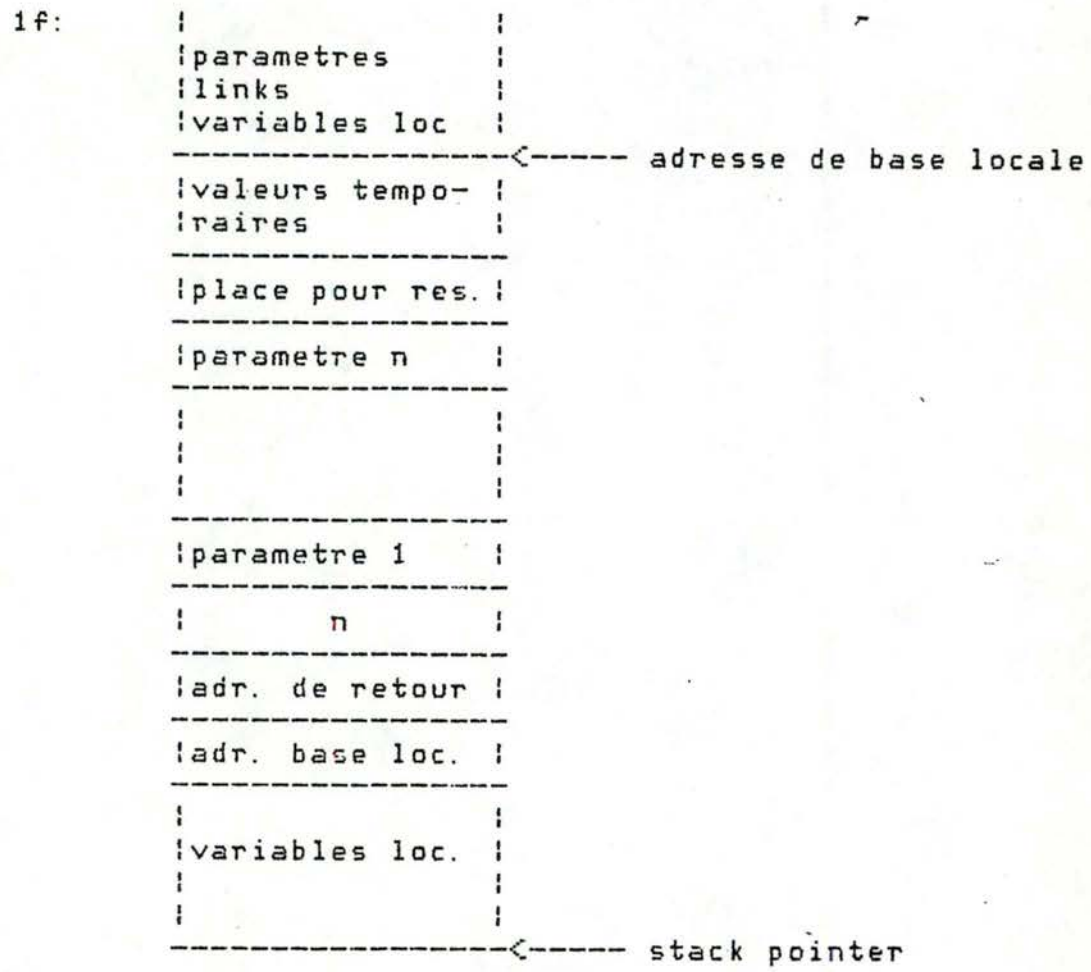
1b:


parametres
links
variables loc
adresse de base locale
valeurs tempo-
raires
place pour res.
stack pointer

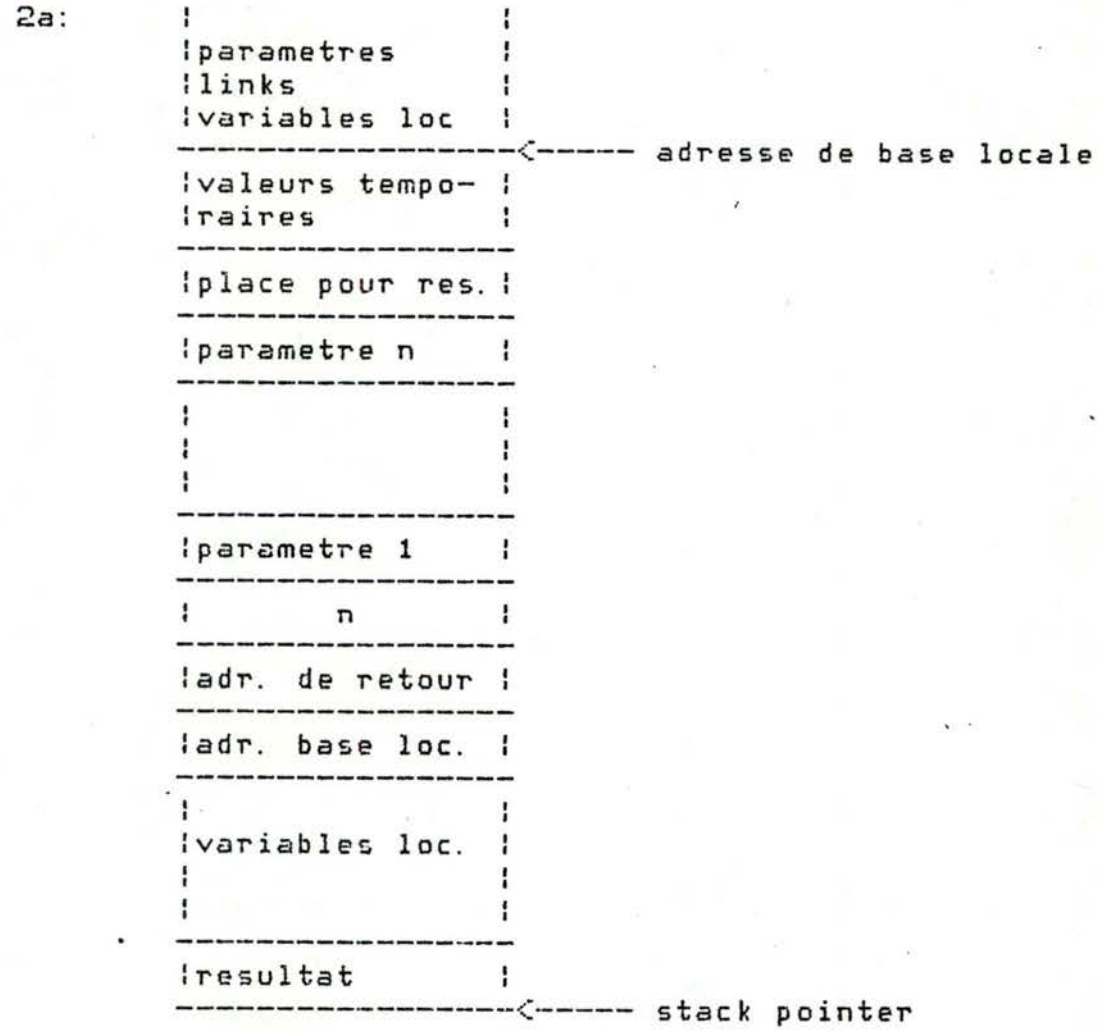
1c:


parametres
links
variables loc.
adresse de base locale
valeurs tempo-
raires
place pour res.
parametre n
parametre 1
n
stack pointer

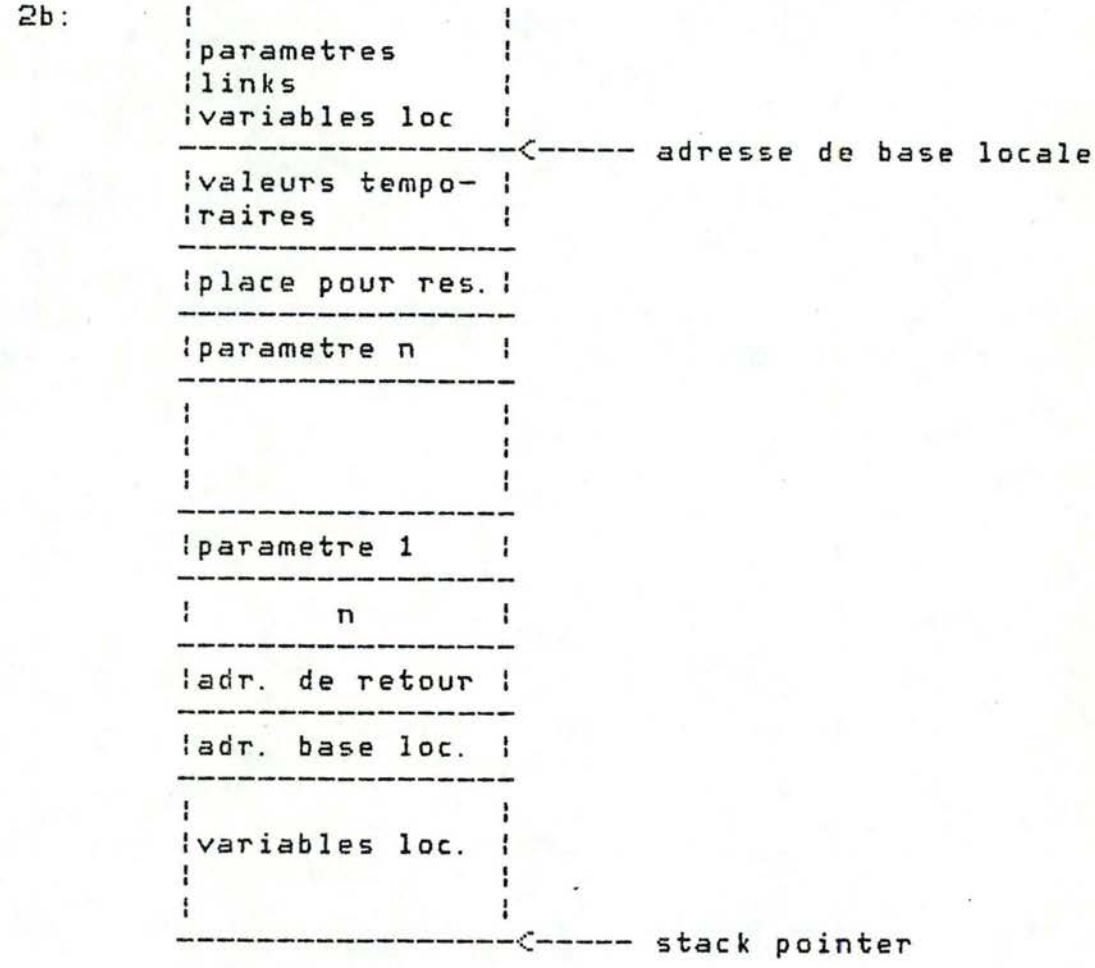

ld:
parametres
links
variables loc.
adresse de base locale
valeurs tempo-
raires
place pour res.
parametre n
parametre 1
n
adr. de retour
stack pointer
ld:
parametres
links
variables loc
adresse de base locale
valeurs tempo-
raires
place pour res.
parametre n
parametre 1
n
adr. de retour
adr. base loc.
stack pointer


lf:
parametres
links
variables loc
adresse de base locale
valeurs tempo-
raires
place pour res.
parametre n
parametre 1
n
adr. de retour
adr. base loc.
variables loc.
stack pointer

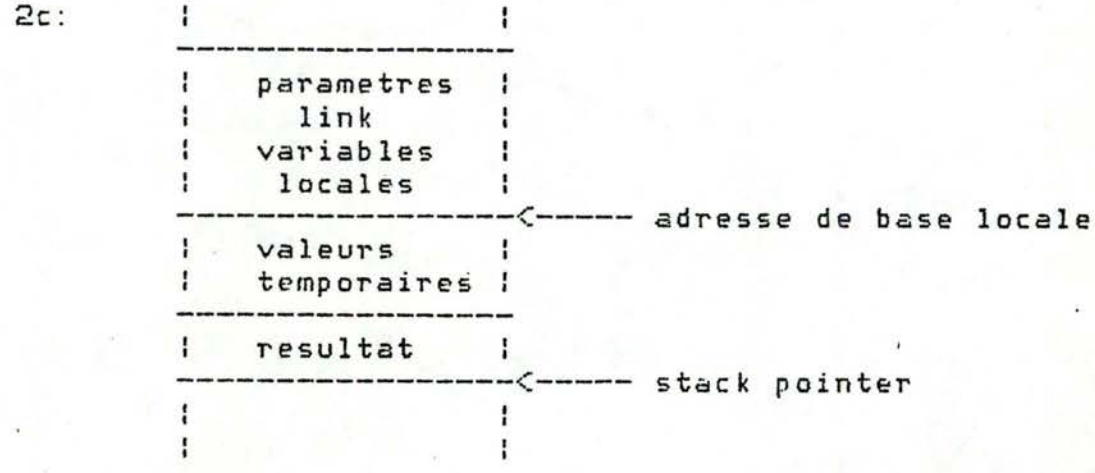
2a:


parametres
links
variables loc
adresse de base locale
valeurs tempo-
raires
place pour res.
parametre n
parametre 1
n
adr. de retour
adr. base loc.
variables loc.
resultat
stack pointer

2b:


parametres
links
variables loc
adresse de base locale
valeurs tempo-
raires
place pour res.
parametre n
parametre 1
n
adr. de retour
adr. base loc.
variables loc.
stack pointer

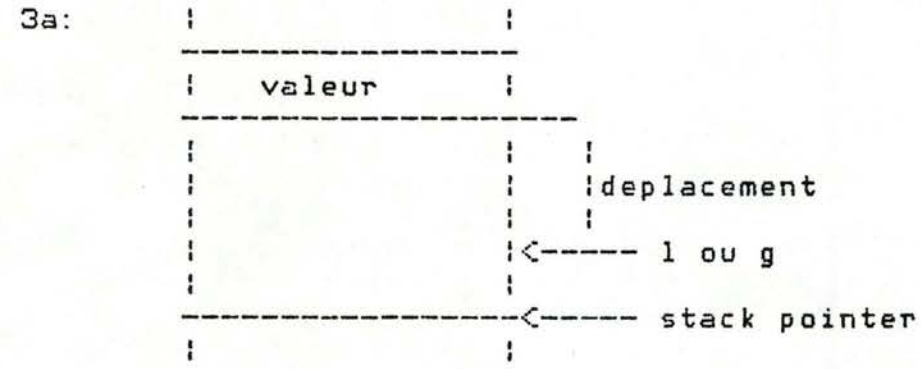
2c:


parametres
link
variables
locales
adresse de base locale
valeurs
temporaires
resultat
stack pointer

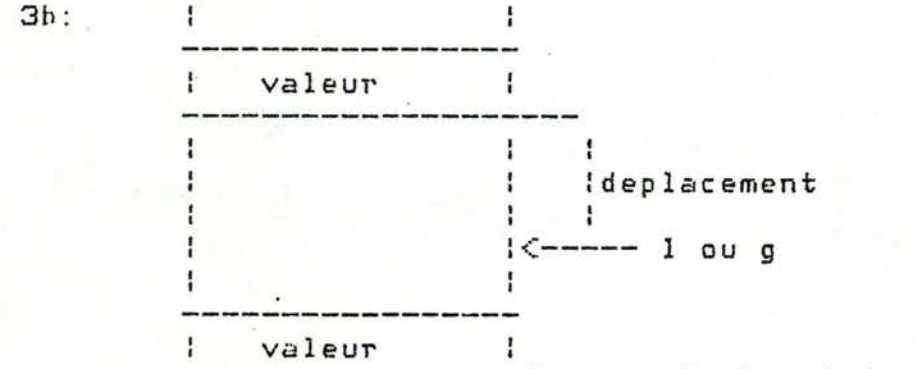
Instructions de manipulation de valeurs

operation	operande	operande sur la pile	pile av	pile ap	fonction
LVLOCI	deplac. % l		3a	3b	placer une valeur entiere sur la pile; deplac. % adresse de base locale l
LVEXTI	deplac. % g		3a 3a	3b 3b	idem LVLOCI mais deplac. % % adresse de base globale
LVLOCC	deplac. % l		3a	3b	idem LVLOCI mais pour des caracteres
LVEXTC	deplac. % g		3a	3b	idem LVEXTI mais pour des caracteres
LVCON	const.		3c	3d	placer une constante sur la pile
INDIRI		adresse	3e	3f	remplacer l'adresse sur la pile par la valeur sur laquelle elle pointe
INDIRC		adresse	3e	3f	idem INDIRI mais pour des caracteres
SVALI		adresse et valeur	3g	3h	changer la valeur entiere sur laquelle pointe l'adr. au sommet de la pile par la valeur entiere egalement sur la pile
SVALC		adresse et valeur	3g	3h	idem SVALI mais pour des caracteres
DUPLI		valeur ent.	3i	3j	duplicier la valeur au sommet de la pile
REMVEI		valeur ent.	3h	31	retirer la valeur entiere au sommet de la pile
LVPARI	deplac. % l		3a	3b	idem LVLOCI mais pour des parametres
LVPARC	deplac. % l		3a	3b	idem LVLOCC mais pour des parametres

3a:


valeur
deplacement
l ou g
stack pointer

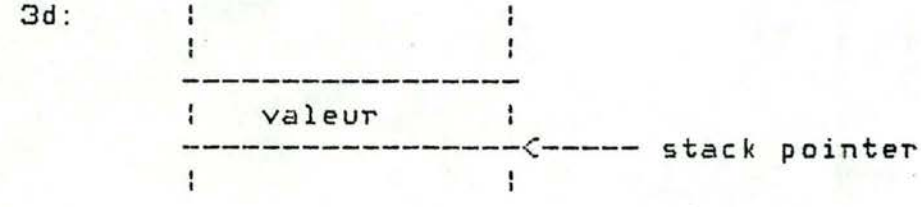
3b:


valeur
deplacement
l ou g
valeur
stack pointer

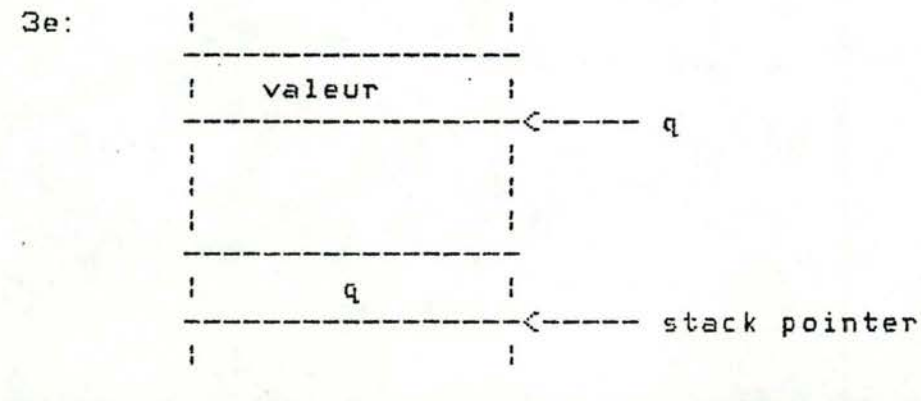
3c:


stack pointer

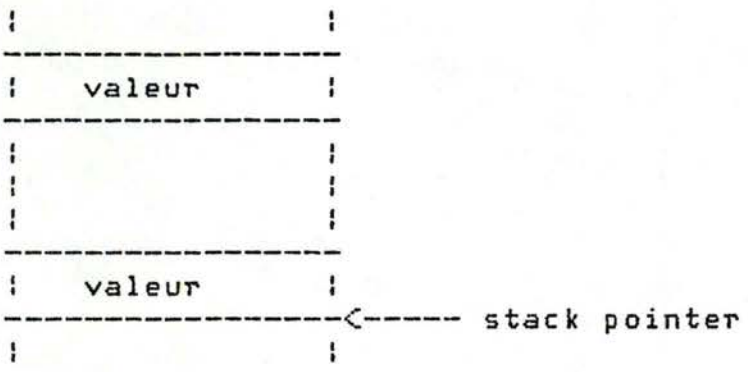
3d:


valeur
stack pointer

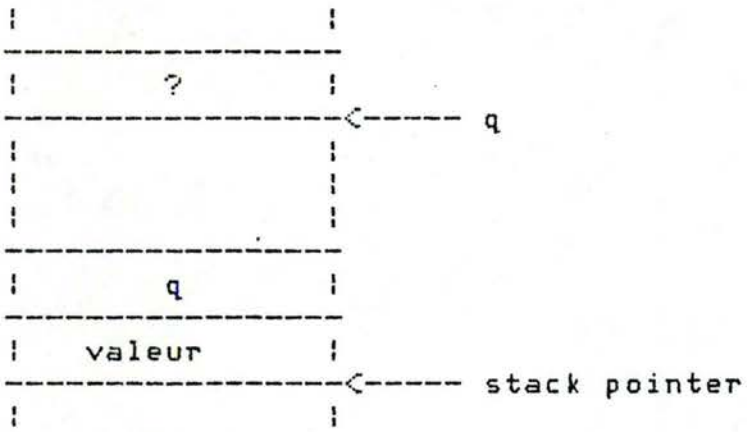
3e:


valeur
q
q
stack pointer

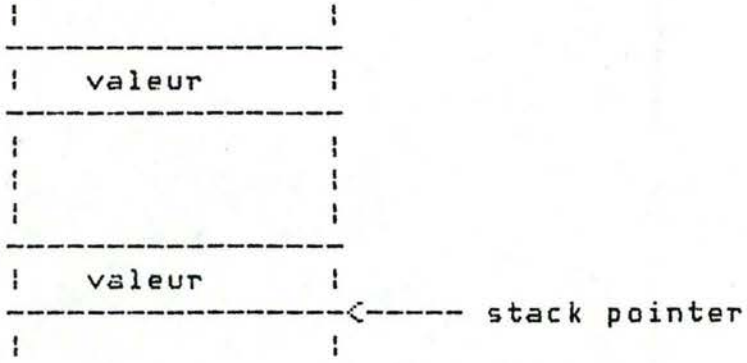
3f:


valeur
valeur
stack pointer

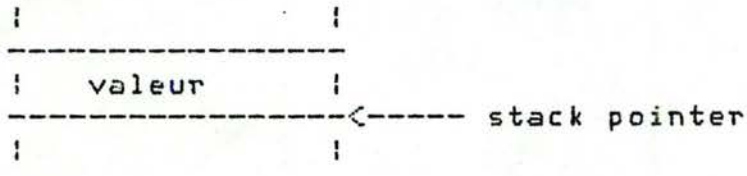
3g:


?
q
q
valeur
stack pointer

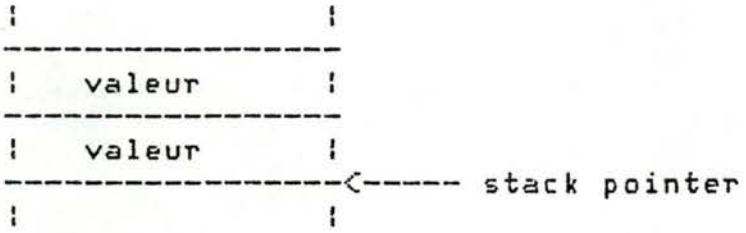
3h:


valeur
valeur
stack pointer

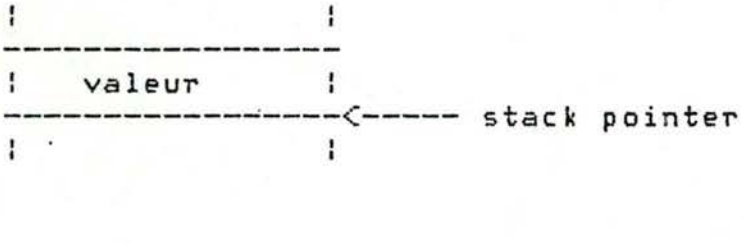
3i:


valeur
stack pointer

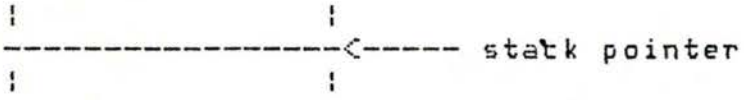
3j:



3k:



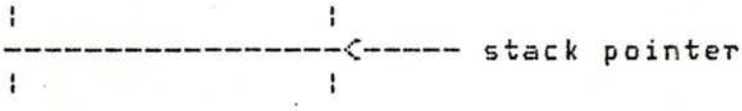
3l:



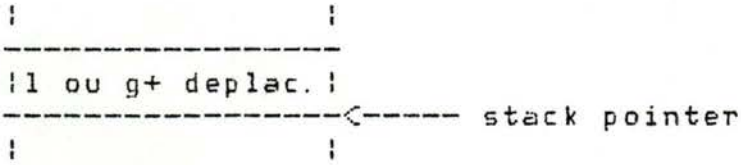
III Instructions de manipulation d'adresses

operation	operande	pile av	pile ap	fonction
LALOC	deplac. % 1	4a	4b	placer l'adresse d'une variable sur la pile(variable locale)
LAPAR	deplac. % 1	4a	4b	idem LALOC mais pour des parametres
LAEXT	deplac. % g	4a	4b	idem LALOC mais deplacement relatif a l'adresse de base globale

4a:



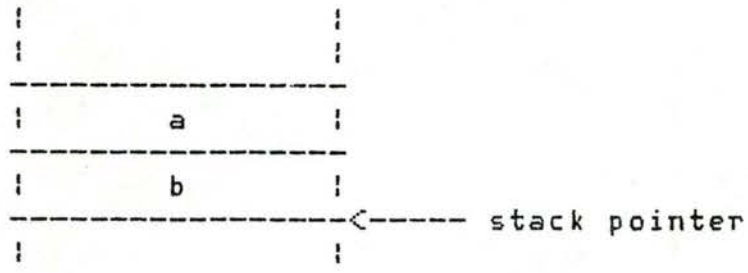
4b:



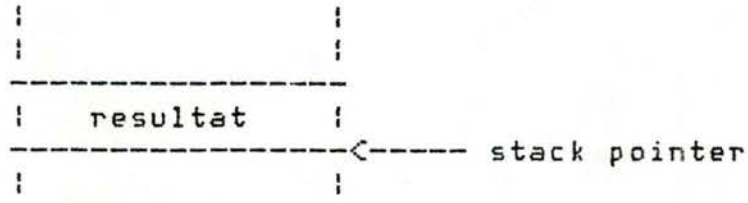
IV Operations binaires

operation	fonction
PLUSI	a+b
SUBTRI	a-b
MULTI	a*b
DIVI	a/b
REMI	a%b
ANDI	a&b
ORI	a|b
EXORI	a^b
RSHFTI	a>>b
LSHFTI	a<<b
EQI	(a==b?1:0)
NEI	(a!=b?1:0)
LEI	(a<=b?1:0)
LTI	(a<b?1:0)
GEI	(a>=b?1:0)
GTI	(a>b?1:0)

pile avant:



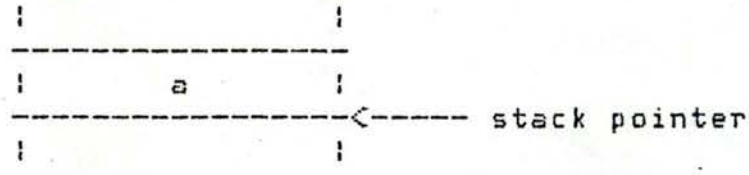
pile apres:



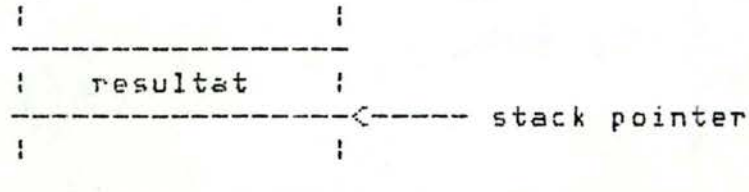
V Operations unaires

operation	fonction
NEGI	-a
COMPI	~a
NOTI	!a

pile avant:



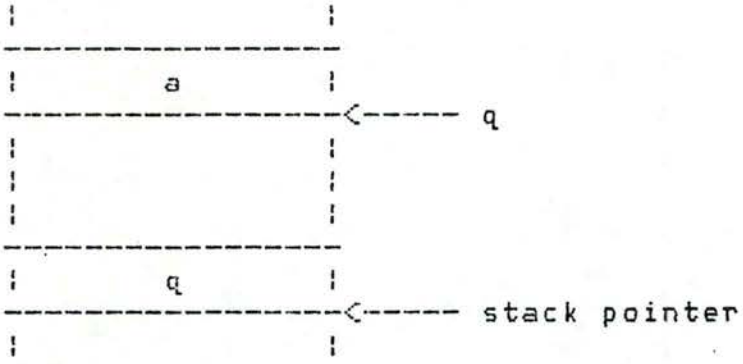
pile apres:



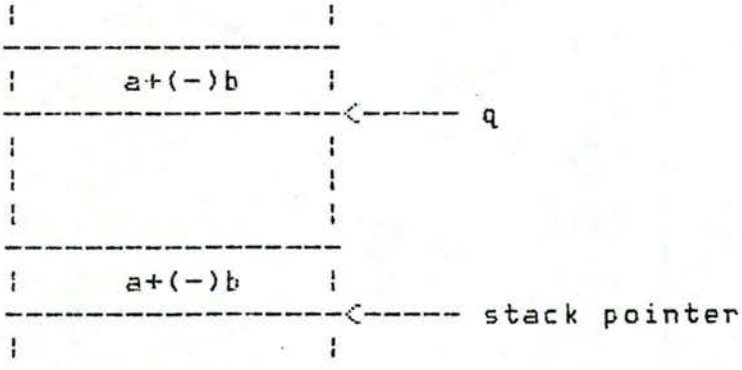
VI Increments et decrements

operation	operande	operande sur la pile	pile av	pile ap	fonction
INCBEF	valeur (b)	adresse	5a	5b	a=a+b resultat sur la pile
DECBEF	valeur (b)	adresse	5a	5b	a=a-b resultat sur la pile
INCAFT	valeur (b)	adresse	5c	5d	a=a+b resultat sur la pile
DECAFT	valeur (b)	adresse	5c	5d	a=a-b resultat sur la pile

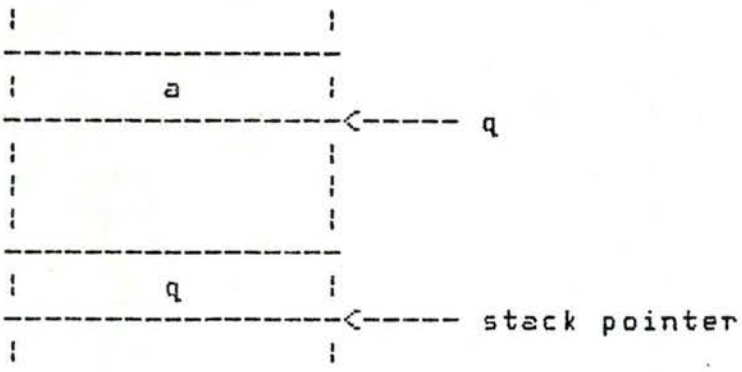
5a:



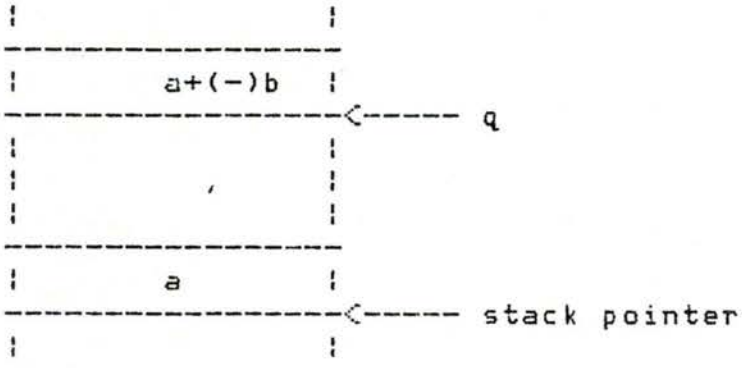
5b:



5c:



5d:



VII Instructions switchs

Les instructions de switch sont au nombre de 3:
-SWTCHD: si les valeurs des differents cas sont proches l'une de l'autre(ecart inferieur a 3)
-SWTCHS: si le nombre de cas est inferieur a 3
-SWTCHH: dans les autres cas

La valeur du switch se trouve au sommet de la pile(sv)

SWTCHD

arguments:

min	plus petite valeur d'un 'case'
range	plus grande - plus petite valeur
def	etiquette pour 'case default'
tabl	table d'etiquettes pour toutes les valeurs de range

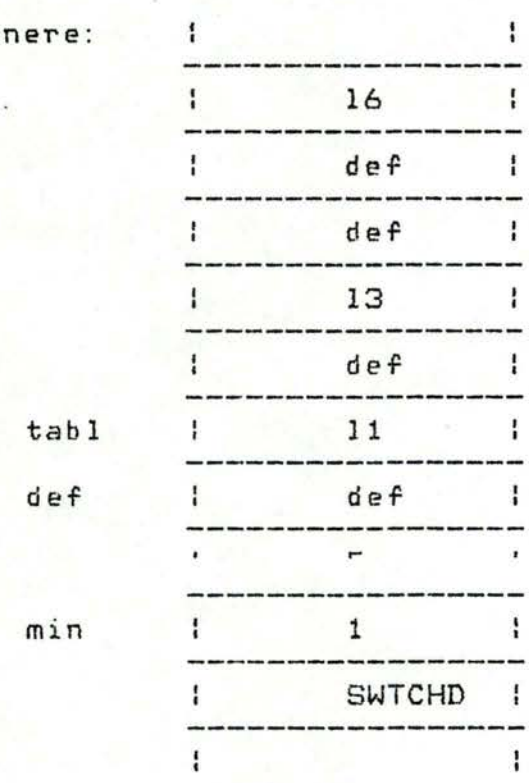
L'effet de l'instruction est le suivant:

```
if(sv-min<0) ! (sv-min>range)
        go to def;
else
        go to tabl[sv-min];
```

Exemple:

```
instruction C: switch(sv) {
                        case 1:......
                        case 3:......
                        case 6:......
                        default:.....
               };
```

code genere:



SWTCHS:
---

arguments:

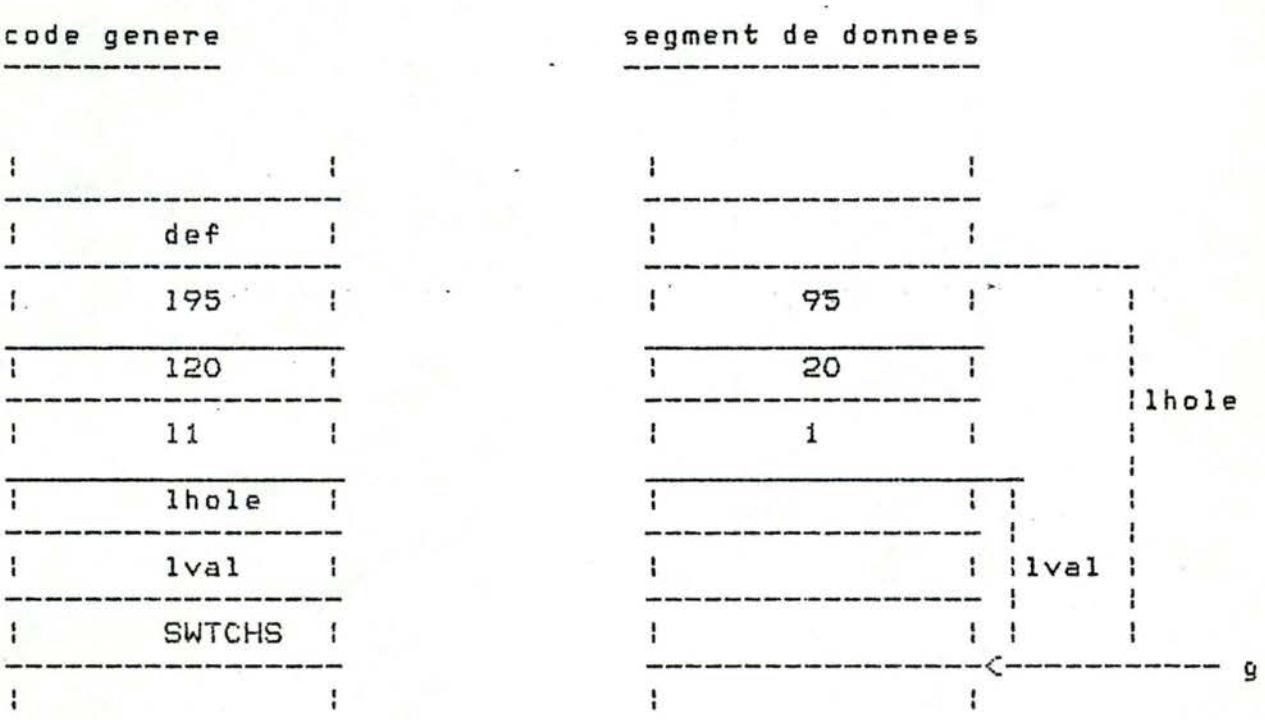
- lval: deplacement relatif a g de la table de case value
- lhole: deplacement relatif a g d'une place pour un entier; cette place suit immediatement la derniere entree dans lval
- tabl: table d'etiquettes dans le meme ordre que les case values et suivie de def

effet de l'instruction:

```
lhole=sv;
i=0;
while(lval[i++]!=sv);
go to tabl[i-1];
```

exemple:

```
switch(sv) {
        case 1:.....
        case 20:....
        case 95:....
        default:....
};
```



SWTCHH:

Pour cette instruction, les case value sont groupes en m classes d'equivalence (relation congruentielle modulo m). On calcule d'abord la classe appropriee(abs(sv%m)). Ensuite une recherche sequentielle est faite a travers les tables comme pour l'instruction SWTCHS.

Pour chaque classe il y a deux tables:

-une table avec les qyotients des case value divises par m

-une table des etiquettes correspondantes; cette table commence avec l'etiquette def

arguments:

m: nombre de sous-tables

lind: deplacement d'une table qui comprend les deplacements relatifs aux tables de quotients

tabl: table des etiquettes

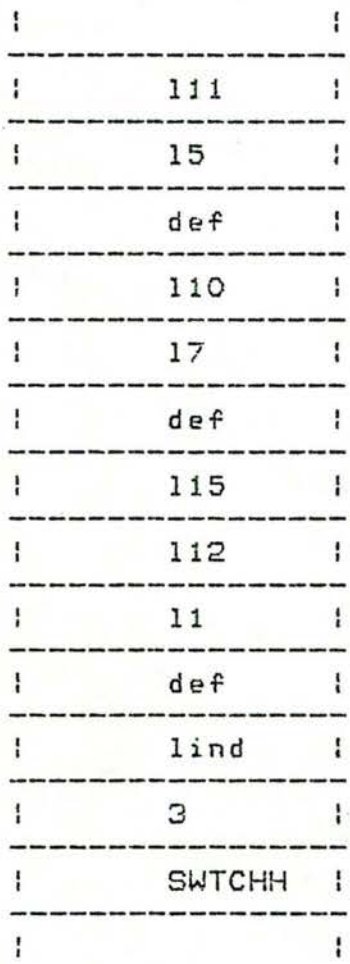
effet de l'instruction:

```
p=lind[sv%m];
q=lind[sv%m+1];
*p=sv;
while(*(--q) != sv);
go to tabl[q-lrem0];
```

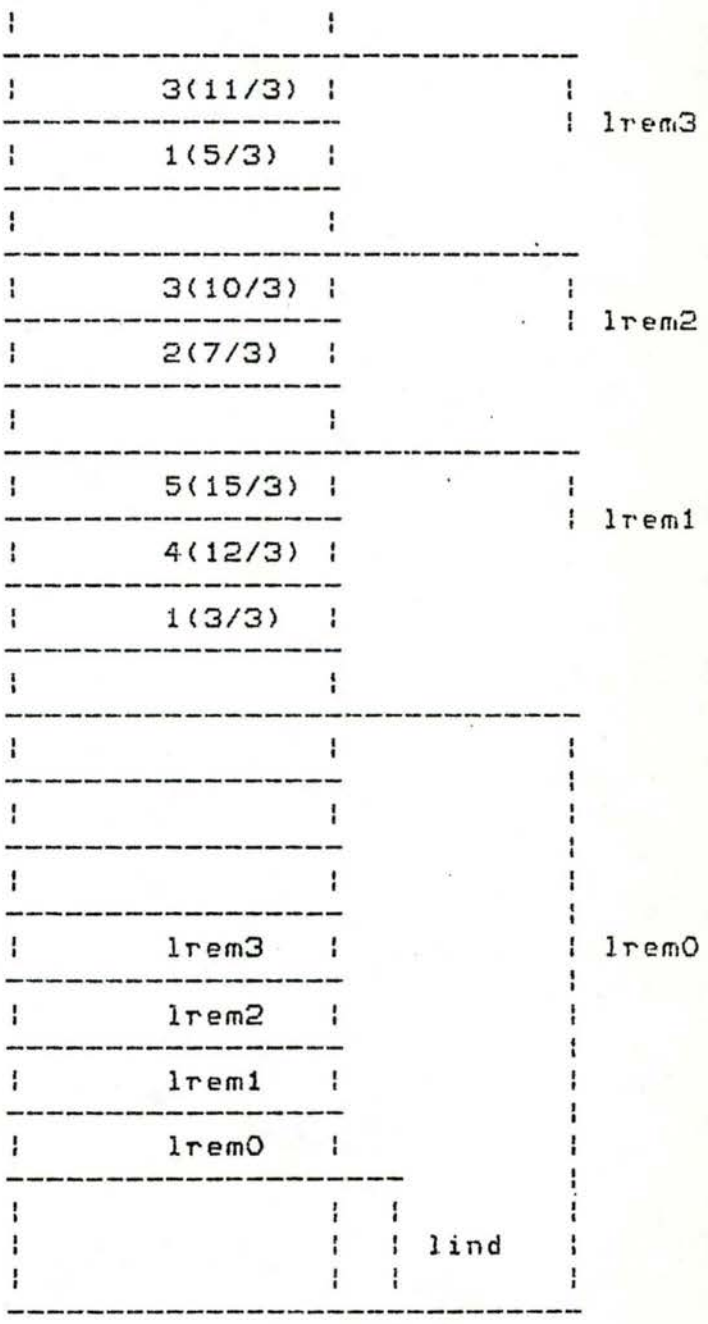
exemple:

```
switch(sv) {
        case 3:.....
        case 5:.....
        case 7:.....
        case 10:....
        case 11:....
        case 12:....
        case 15:....
};
```

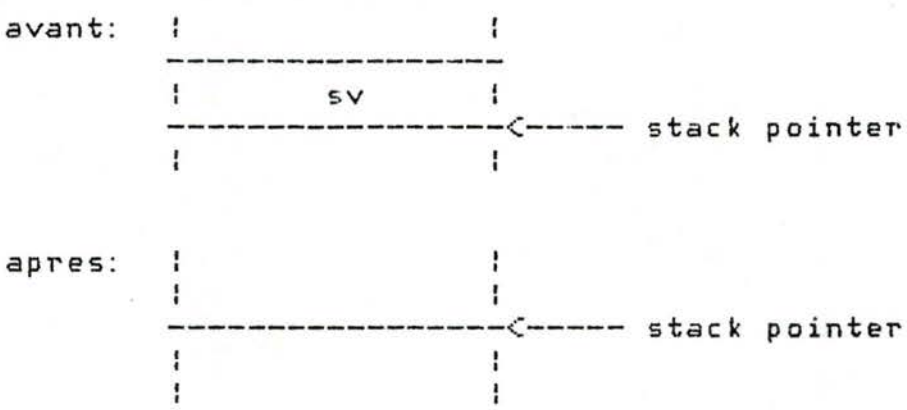
code genere



segment de donnees

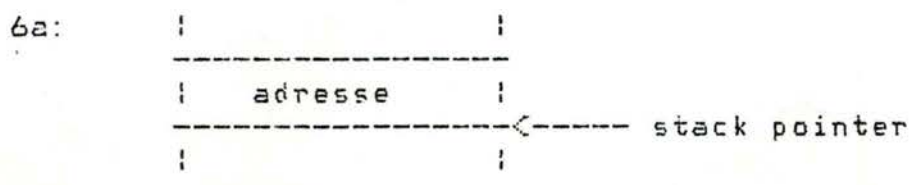


effet sur la pile(dans les 3 cas):

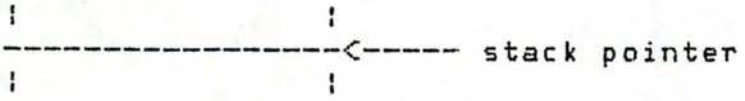


VIII Instructions de branchements

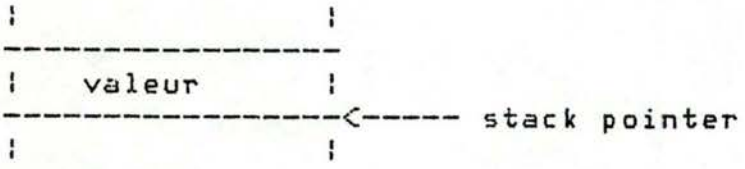
operation	operande	operande sur la pile	pile av	pile ap	fonction
JUMP	label				branchement incondionnel
JUMPI		adresse	6a	6b	branchement inconditionnel a l'adr. sur la pile
JUMPT	label	valeur	6c	6d	branchement a label si l'element sur la pile != 0
JUMPF	label	valeur	6c	6d	idem JUMPT mais branch. si element sur la pile==0
JEQ	label	a,b entiers	6e	6f	branchement a label si a==b
JNE	label	a,b entiers	6e	6f	branchement a label si a!=b
JLT	label	a,b entiers	6e	6f	branchement a label si a<b
JLE	label	a,b entiers	6e	6f	branchement a label si a<=b
JGT	label	a,b entiers	6e	6f	branchement a label si a>b
JGE	label	a,b entiers	6e	6f	branchement a label si a>=b



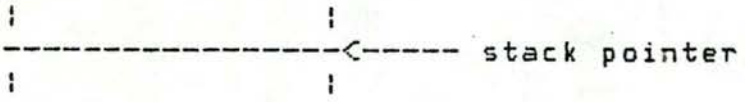
6b:


stack pointer

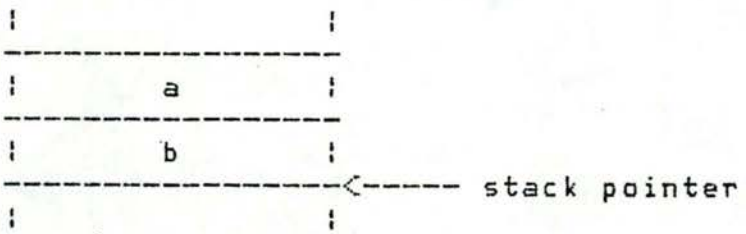
6c:


valeur
stack pointer

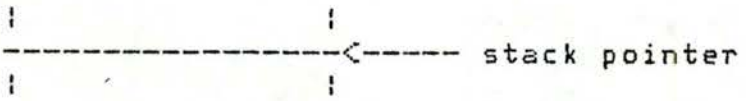
6d:


stack pointer

6e:


a
b
stack pointer

6f:


stack pointer

ANNEXE 2

MOTOROLA 6800

A. JEU D'INSTRUCTIONS STANDARD
B. JEU D'INSTRUCTIONS CALM
C. DESCRIPTION UNIASS
D. PROGRAMMES DE TRADUCTION ET D'OPTIMISATION
   - motorola.1
   - motorola.2

A. JEU D'INSTRUCTION STANDARD DU MOTOROLA 6800

APPENDIX A

6800 INSTRUCTION SET

This table presents the entire instruction set of the 6800 CPU. The first column contains the mnemonic used for each instruction. The machine code, presented as hexadecimal digits, is given in the second column. For the instructions that use one or more of the possible addressing modes, the third column indicates the mode for the machine code of that row. The fourth and fifth columns indicate the number of bytes required for the instruction cycles and the number of machine cycles, respectively. The final column indicates the page in chapter one in which the instruction is defined.

MNEMONIC	MACHINE CODE	ADDRESSING MODE	NO. OF BYTES	NO. OF CYCLES	PAGE REF.
ABA	1B		1	2	1-23
ADCA	89	IMMEDIATE	2	2	1-22
ADCA	99	DIRECT	2	3	1-22
ADCA	A9	INDEXED	2	5	1-22
ADCA	B9	EXTENDED	3	4	1-22
ADCB	C9	IMMEDIATE	2	2	1-22
ADCB	D9	DIRECT	2	3	1-22
ADCB	E9	INDEXED	2	5	1-22
ADCB	F9	EXTENDED	3	4	1-22
ADDA	8B	IMMEDIATE	2	2	1-22
ADDA	9B	DIRECT	2	3	1-22
ADDA	AB	INDEXED	2	5	1-22
ADDA	BB	EXTENDED	3	4	1-22
ADDB	CB	IMMEDIATE	2	2	1-22
ADDB	DB	DIRECT	2	3	1-22
ADDB	EB	INDEXED	2	5	1-22
ADDB	FB	EXTENDED	3	4	1-22
ANDA	84	IMMEDIATE	2	2	1-28
ANDA	94	DIRECT	2	3	1-28
ANDA	A4	INDEXED	2	5	1-28
ANDA	B4	EXTENDED	3	4	1-28
ANDB	C4	IMMEDIATE	2	2	1-28
ANDB	D4	DIRECT	2	3	1-28
ANDB	E4	INDEXED	2	5	1-28
ANDB	F4	EXTENDED	3	4	1-28
ASL	68	INDEXED	2	7	1-34
ASL	78	EXTENDED	3	6	1-34
ASLA	48		1	2	1-34
ASLB	58		1	2	1-34
ASR	67	INDEXED	2	7	1-35

ASR	77	EXTENDED	3	6	1-35
ASRA	47		1	2	1-35
ASRB	57		1	2	1-35
BCC	24	RELATIVE	2	4	1-39
BCS	25	RELATIVE	2	4	1-39
BEQ	27	RELATIVE	2	4	1-39
BGE	2C	RELATIVE	2	4	1-40
BGT	2E	RELATIVE	2	4	1-40
BHI	22	RELATIVE	2	4	1-39
BITA	85	IMMEDIATE	2	2	1-30
BITA	95	DIRECT	2	3	1-30
BITA	A5	INDEXED	2	5	1-30
BITA	B5	EXTENDED	3	4	1-30
BITB	C5	IMMEDIATE	2	2	1-30
BITB	D5	DIRECT	2	3	1-30
BITB	E5	INDEXED	2	5	1-30
BITB	F5	EXTENDED	3	4	1-30
BLE	2F	RELATIVE	2	4	1-40
BLS	23	RELATIVE	2	4	1-39
BLT	2D	RELATIVE	2	4	1-40
BMI	2B	RELATIVE	2	4	1-39
BNE	26	RELATIVE	2	4	1-39
BPL	2A	RELATIVE	2	4	1-39
BRA	20	RELATIVE	2	4	1-38
BSR	8D	RELATIVE	2	8	1-42
BVC	28	RELATIVE	2	4	1-39
BVS	29	RELATIVE	2	4	1-39
CBA	11		1	2	1-26
CLC	0C		1	2	1-31
CLI	0E		1	2	1-32
CLR	6F	INDEXED	2	7	1-20
CLR	7F	EXTENDED	3	6	1-20
CLRA	4F		1	2	1-15
CLRB	5F		1	2	1-15
CLV	0A		1	2	1-32
CMPA	81	IMMEDIATE	2	2	1-26
CMPA	91	DIRECT	2	3	1-26
CMPA	A1	INDEXED	2	5	1-26
CMPA	B1	EXTENDED	3	4	1-26
CMPB	C1	IMMEDIATE	2	2	1-26
CMPB	D1	DIRECT	2	3	1-26
CMPB	E1	INDEXED	2	5	1-26
CMPB	F1	EXTENDED	3	4	1-26
COM	63	INDEXED	2	7	1-21
COM	73	EXTENDED	3	6	1-21
COMA	43		1	2	1-16
COMB	53		1	2	1-16
CPX	8C	IMMEDIATE	3	3	1-26
CPX	9C	DIRECT	2	4	1-26
CPX	AC	INDEXED	2	6	1-26
CPX	BC	EXTENDED	3	5	1-26
DAA	19		1	2	1-23
DEC	6A	INDEXED	2	7	1-21
DEC	7A	EXTENDED	3	6	1-21
DECA	4A		1	2	1-15
DECB	5A		1	2	1-15
DES	34		1	4	1-19
DEX	09		1	4	1-18
EORA	88	IMMEDIATE	2	2	1-30
EORA	98	DIRECT	2	3	1-30
EORA	A8	INDEXED	2	5	1-30
EORA	B8	EXTENDED	3	4	1-30
EORB	C8	IMMEDIATE	2	2	1-30
EORB	D8	DIRECT	2	3	1-30
EORB	E8	INDEXED	2	5	1-30
EORB	F8	EXTENDED	3	4	1-30
INC	6C	INDEXED	2	7	1-20
INC	7C	EXTENDED	3	6	1-20
INCA	4C		1	2	1-15
INCB	5C		1	2	1-15
INS	31		1	4	1-19
INX	08		1	4	1-17
JMP	6E	INDEXED	2	4	1-41
JMP	7E	EXTENDED	3	3	1-41
JSR	AD	INDEXED	2	8	1-42
JSR	BD	EXTENDED	3	9	1-42
LDAA	86	IMMEDIATE	2	2	1-13
LDAA	96	DIRECT	2	3	1-13
LDAA	A6	INDEXED	2	5	1-13
LDAA	B6	EXTENDED	3	4	1-13
LDAB	C6	IMMEDIATE	2	2	1-13
LDAB	D6	DIRECT	2	3	1-13
LDAB	E6	INDEXED	2	5	1-13
LDAB	F6	EXTENDED	3	4	1-13
LDS	8E	IMMEDIATE	3	3	1-18
LDS	9E	DIRECT	2	4	1-18
LDS	AE	INDEXED	2	6	1-18
LDS	BE	EXTENDED	3	5	1-18
LDX	CE	IMMEDIATE	3	3	1-17
LDX	DE	DIRECT	2	4	1-17
LDX	EE	INDEXED	2	6	1-17
LDX	FE	EXTENDED	3	5	1-17
LSR	64	INDEXED	2	7	1-35
LSR	74	EXTENDED	3	6	1-35
LSRA	44		1	2	1-35

LSRB	54		1	2	1-35
NEG	60	INDEXED	2	7	1-21
NEG	70	EXTENDED	3	6	1-21
NEGA	40		1	2	1-16
NEGB	50		1	2	1-16
NOP	01		1	2	1-37
ORAA	8A	IMMEDIATE	2	2	1-29
ORAA	9A	DIRECT	2	3	1-29
ORAA	AA	INDEXED	2	5	1-29
ORAA	BA	EXTENDED	3	4	1-29
ORAB	CA	IMMEDIATE	2	2	1-29
ORAB	DA	DIRECT	2	3	1-29
ORAB	EA	INDEXED	2	5	1-29
ORAB	FA	EXTENDED	3	4	1-29
PSHA	36		1	4	1-14
PSHB	37		1	4	1-14
PULA	32		1	4	1-14
PULB	33		1	4	1-14
ROL	69	INDEXED	2	7	1-36
ROL	79	EXTENDED	3	6	1-36
ROLA	49		1	2	1-36
ROLB	59		1	2	1-36
ROR	66	INDEXED	2	7	1-36
ROR	76	EXTENDED	3	6	1-36
RORA	46		1	2	1-36
RORB	56		1	2	1-36
RTI	3B		1	10	1-44
RTS	39		1	5	1-43
SBA	10		1	2	1-25
SBCA	82	IMMEDIATE	2	2	1-24
SBCA	92	DIRECT	2	3	1-24
SBCA	A2	INDEXED	2	5	1-24
SBCA	B2	EXTENDED	3	4	1-24
SBCB	C2	IMMEDIATE	2	2	1-24
SBCB	D2	DIRECT	2	3	1-24
SBCB	E2	INDEXED	2	5	1-24
SBCB	F2	EXTENDED	3	4	1-24
SEC	0D		1	2	1-31
SEI	0F		1	2	1-32
SEV	0B		1	2	1-32
STAA	97	DIRECT	2	4	1-14
STAA	A7	INDEXED	2	6	1-14
STAA	B7	EXTENDED	3	5	1-14
STAB	D7	DIRECT	2	4	1-14
STAB	E7	INDEXED	2	6	1-14
STAB	F7	EXTENDED	3	5	1-14
STS	9F	DIRECT	2	5	1-18
STS	AF	INDEXED	2	7	1-18
STS	BF	EXTENDED	3	6	1-18
STX	DF	DIRECT	2	5	1-17
STX	EF	INDEXED	2	7	1-17
STX	FF	EXTENDED	3	6	1-17
SUBA	80	IMMEDIATE	2	2	1-24
SUBA	90	DIRECT	2	3	1-24
SUBA	A0	INDEXED	2	5	1-24
SUBA	B0	EXTENDED	3	4	1-24
SUBB	C0	IMMEDIATE	2	2	1-24
SUBB	D0	DIRECT	2	3	1-24
SUBB	E0	INDEXED	2	5	1-24
SUBB	F0	EXTENDED	3	4	1-24
SWI	3F		1	12	1-44
TAB	16		1	2	1-13
TAP	06		1	2	1-33
TBA	17		1	2	1-13
TPA	07		1	2	1-33
TST	6D	INDEXED	2	7	1-27
TST	7D	EXTENDED	3	6	1-27
TSTA	4D		1	2	1-27
TSTB	5D		1	2	1-27
TSX	30		1	4	1-19
TXS	35		1	4	1-19
WAI	3E		1	9	1-43

B. JEU D'INSTRUCTION CALM DU MOTOROLA 6800

0	
1	
2	NOP
3	
4	
5	
6	LOAD F, A
7	LOAD A, F
10	INC IX
11	DEC IX
12	CLRV
13	SETV
14	CLRX
15	SETC
16	IOF
17	ION
A0	SUB A, B
A1	COMP A, B
A2	
A3	
A4	
A5	
A6	LOAD B, A
A7	LOAD A, B
B0	
B1	DAA
B2	
B3	ADD A, B
B4	
B5	
B6	
B7	
C0	JUMP , +n'
C1	
C2	JUMP, NU , +n'
C3	JUMP, LS , +n'
C4	JUMP, GT , +n'
C5	JUMP, VS , +n'
C6	JUMP, NE , +n'
C7	JUMP, EQ , +n'
C8	JUMP, VC , +n'
C9	JUMP, VS , +n'
CA	JUMP, PL , +n'
CB	JUMP, MI , +n'
CC	JUMP, GE , +n'
CD	JUMP, LT , +n'
CE	JUMP, GT , +n'
CF	JUMP, LE , +n'
D0	INC IX, SP
D1	INC SP
D2	POP A
D3	POP B
D4	DEC SP
D5	DEC SP, 1Y
D6	PUSH A
D7	PUSH B

200	SUB A, #n
201	COMP A, #n
202	SUBC A, #n
203	
204	AND A, #n
205	BIT A, #n
206	LOAD A, #n
207	
210	XOR A, #n
211	ADDC A, #n
212	OR A, #n
213	ADD A, #n
214	COMP IX, #m
215	CALL . +n'
216	LOAD SP, #m
217	
220	SUB A, n
221	COMP A, n
222	SUBC A, n
223	
224	AND A, n
225	BIT A, n
226	LOAD A, n
227	LOAD n, A
230	XOR A, n
231	ADDC A, n
232	OR A, n
233	ADD A, n
234	COMP IX, n
235	
236	LOAD SP, n
237	LOAD n, SP
240	SUB A, (IX)+n
241	COMP A, (IX)+n
242	SUBC A, (IX)+n
243	
244	AND A, (IX)+n
245	BIT A, (IX)+n
246	LOAD A, (IX)+n
247	LOAD (IX)+n, A
250	XOR A, (IX)+n
251	ADDC A, (IX)+n
252	OR A, (IX)+n
253	ADD A, (IX)+n
254	COMP IX, (IX)+n
255	
256	LOAD SP, (IX)+n
257	LOAD (IX)+n, SP
260	SUB A, m
261	COMP A, m
262	SUBC A, m
263	
264	AND A, m
265	BIT A, m
266	LOAD A, m
267	LOAD m, A

70	
71	RET
72	
73	RTI
74	
75	
76	WAIT
77	TRAP
100	NEG A
101	
102	
103	CPL A
104	SRC A
105	
106	RRC A
107	ASR A
110	SLC A
111	RLC A
112	DEC A
113	
114	INC A
115	TEST A
116	
117	CLR A
120	NEG B
121	
122	
123	CPL B
124	SRC B
125	
126	RRC B
127	ASR B
130	SLC B
131	RLC B
132	DEC B
133	
134	INC B
135	TEST B
136	
137	CLR B
140	NEG (IX)+n
141	
142	
143	CPL (IX)+n
144	SRC (IX)+n
145	
146	RRC (IX)+n
147	ASR (IX)+n
150	SLC (IX)+n
151	RLC (IX)+n
152	DEC (IX)+n
153	
154	INC (IX)+n
155	TEST (IX)+n
156	
157	CLR (IX)+n

270	XOR A,m
271	ADDC A,m
272	OR A,m
273	ADD A,m
274	COMP IX,m
275	
276	LOAD SP,m
277	LOAD m,SP
300	SUB B,#n
301	COMP B,#n
302	SUBC B,#n
303	
304	AND B,#n
305	BIT B,#n
306	LOAD B,#n
307	
310	XOR B,#n
311	ADDC B,#n
312	OR B,#n
313	ADD B,#n
314	
315	
316	LOAD IX,#nn
317	
320	SUB B,n
321	COMP B,n
322	SUBC B,n
323	
324	AND B,n
325	BIT B,n
326	LOAD B,n
327	LOAD n,B
330	XOR B,n
331	ADDC B,n
332	OR B,n
333	ADD B,n
334	
335	
336	LOAD IX,n
337	LOAD n,IX
340	SUB B,(IX)+n
341	COMP B,(IX)+n
342	SUBC B,(IX)+n
343	
344	AND B,(IX)+n
345	BIT B,(IX)+n
346	LOAD B,(IX)+n
347	LOAD (IX)+n,B
350	XOR B,(IX)+n
351	ADDC B,(IX)+n
352	OR B,(IX)+n
353	ADD B,(IX)+n
354	
355	
356	LOAD IX,(IX)+n
357	LOAD (IX)+n,IX

160	NEG m	360	SUB B,m
161		361	COMP B,m
162		362	SUBC B,m
163	CPL m	363	
164	SRC m	364	AND B,m
165		365	BIT B,m
166	RRC m	366	LOAD B,m
167	ASR m	367	LOAD m,B
170	SLC m	370	XOR B,m
171	RLC m	371	ADDC B,m
172	DEC m	372	OR B,m
173		373	ADD B,m
174	INC m	374	
175	TEST m	375	
176		376	LOAD IX,m
177	CLR m	377	LOAD m,IX

C. DESCRIPTION UNIASS DU MOTOROLA 6800

```
                .LIST     L-

                ;MOTOROLA  6800
                ;==============

    ;DESCRIPTION FOR THE UNIVERSAL CROSS-ASSEMBLER
    ;=============================================

    ;VERSION 1.1          DECEMBER 1977          AMS
    ;-----------          -------------          ---


MPNAME ' MOTOROLA 6800 MICRO-PROCESSOR'
DECOMP LOW,HIGH


\/  GLOBAL DEFINITIONS:
    ==================


-BIT AND 16-BIT NUMBERS (POSITIVE OR NEGATIVE):
ODE M1 =  V1:=0-200
EST S  =  V>=V1
ODE M2 =  V1:=0-100000
AXI N  =  :M1, ?S
AXI M  =  :M2, ?S

PLITTING A 16-BIT NUMBER INTO 2 BYTES:
IGH BYTE FIRST, LOW BYTE NEXT; AND PLACING THEM
N BYTES 2 AND 3 OF AN INSTRUCTION:
ODE LOWB =  V1:=V&377          ;V1:=LOW BYTE
ODE HB1  =  V2:=V&177400
ODE HB2  =  V3:=V2->8.         ;V3:=HIGH BYTE
ODE B2V3 =  B2:=V3
ODE B3V1 =  B3:=V1
AXI W2   =  :LOWB, :HB1, :HB2, :B2V3, :B3V1

EST TO DETERMINE WHETHER AN ADDRESS IS IN RELATIVE RANGE:
ODE VMPC   =  V1:=V-PC
ODE NPM2   =  V1:=V1-2
ODE M200   =  V2:=0-200
EST LE177  =  V1<=177
EST GEM200 =  V1>=V2
AXI REL    =  :VMPC, :NPM2, :M200, ?LE177, ?GEM200


/   INSTRUCTIONS:
    ============


OAD INSTRUCTIONS:
-----------------

EGISTERS:
NU R    = ('A'=0,'B'=100,'IX'=110,'SP'=10)

EGISTER TESTS:
ST A =  V=0
ST B =  V=100
```

```
I 'LOAD'_'F'_'A'B1(6)                                    ;LOAD F,A
I 'LOAD'_'(IX)+'E(8.)B2(V)_(R)B1(V)B1(247)               ;LOAD (IX)+N,R
I 'LOAD'_(R)B1(V)_'(IX)+'E(8.)B2(V)B1(246)               ;LOAD R,(IX)+N
I 'LOAD'_(R)B1(V)_'#'?A S(9.)$N B2(V)B1(206)             ;LOAD A,#N
I 'LOAD'_(R)B1(V)_'#'?B S(9.)$N B2(V)B1(206)             ;LOAD B,#N
I 'LOAD'_(R)B1(V)_'#'S(15.)$M $W2 B1(206)                ;LOAD IX/SP,#M
I 'LOAD'_(R)B1(V)_?A 'B' B1(27)                          ;LOAD A,B
I 'LOAD'_(R)B1(V)_?A 'F' B1(7)                           ;LOAD A,F
I 'LOAD'_(R)B1(V)_?A E(8.)B2(V)B1(226)                   ;LOAD A,N
I 'LOAD'_(R)B1(V)_?A E(14.)$W2 B1(266)                   ;LOAD A,M
I 'LOAD'_(R)B1(V)_?B 'A' :ZERO1 B1(26)                   ;LOAD B,A
I 'LOAD'_(R)B1(V)_?B E(8.)B2(V)B1(226)                   ;LOAD B,N
I 'LOAD'_(R)B1(V)_?B E(14.)$W2 B1(266)                   ;LOAD B,M
I 'LOAD'_(R)B1(V)_E(8.)B2(V)B1(226)                      ;LOAD IX/SP,N
I 'LOAD'_(R)B1(V)_E(14.)$W2 B1(266)                      ;LOAD IX/SP,M
I 'LOAD'_E(8.)B2(V)_(R)B1(V)B1(227)                      ;LOAD N,R
I 'LOAD'_E(14.) $W2_(R)B1(V)B1(267)                      ;LOAD M,R


ARITHMETIC INSTRUCTIONS:
------------------------

MNEMONICS:
MENU ARITH1 = ('ADD'=13,'SUB'=0,'COMP'=1)
MENU ARITH2 = ('ADDC'=211,'SUBC'=202,'AND'=204,'OR'=212,'XOR'=210,'BIT'=205)

REGISTERS:
MENU R1    = ('A'=0,'B'=100)

MNEMONIC TEST:
TEST C    =  V=1

 (ARITH2)B1(V)_(R1)B1(V)_'(IX)+'E(8.)B2(V)B1(40)    ;ARITH2 R,(IX)+N
 (ARITH2)B1(V)_(R1)B1(V)_'#'S(9.)$N B2(V)           ;ARITH2 R,#N
 (ARITH2)B1(V)_(R1)B1(V)_E(8.)B2(V)B1(20)           ;ARITH2 R,N
 (ARITH2)B1(V)_(R1)B1(V)_E(14.)$W2 B1(60)           ;ARITH2 R,M
 (ARITH1)B1(V)_(R1)B1(V)_'(IX)+'E(8.)B2(V)B1(240)   ;ARITH1 R,(IX)+N
 (ARITH1)B1(V)_(R1)B1(V)_'#'S(9.)$N B2(V)B1(200)    ;ARITH1 R,#N
 (ARITH1)B1(V)_(R1)B1(V)_?A 'B' B1(20)              ;ARITH1 A,B
 (ARITH1)B1(V)_(R1)B1(V)_?A E(8.)B2(V)B1(220)       ;ARITH1 A,N
 (ARITH1)B1(V)_(R1)B1(V)_?A E(14.)$W2 B1(260)       ;ARITH1 A,M
 (ARITH1)B1(V)_(R1)B1(V)_E(8.)B2(V)B1(220)          ;ARITH1 B,N
 (ARITH1)B1(V)_(R1)B1(V)_E(14.)$W2 B1(260)          ;ARITH1 B,M
 (ARITH1)B1(V)_?C'IX':ZERO1 B1(214)_'(IX)+'E(8.)B2(V)B1(40) ;COMP IX,(IX)+N
 (ARITH1)B1(V)_?C'IX':ZERO1 B1(214)_'#'S(15.)$M $W2         ;COMP IX,#M
 (ARITH1)B1(V)_?C'IX':ZERO1 B1(214)_E(8.)B2(V)B1(20)        ;COMP IX,N
 (ARITH1)B1(V)_?C'IX':ZERO1 B1(214)_E(14.)$W2 B1(60)        ;COMP IX,M


DECIMAL ADJUST:
---------------

 'DAA'_'A'B1(31)                                    ;DAA A
 'DAA'_B1(31)                                       ;DAA
 'DAA'B1(31)                                        ;DAA


INCREMENT/DECREMENT INSTRUCTIONS:
---------------------------------
```

```
REGISTERS:
MENU AB    = ('A'=0, 'B'=20)

I 'INC'_'IX'_'SP'B1(60)                                    ;INC IX,SP
I 'INC'_'IX'_B1(10)                                        ;INC IX
I 'INC'_'IX'B1(10)                                         ;INC IX
I 'INC'_'SP'B1(61)                                         ;INC SP
I 'INC'_(AB)B1(114)B1(V)                                   ;INC A/B
I 'INC'_'(IX)+'B1(154)E(8.)B2(V)                           ;INC (IX)+N
I 'INC'_E(14.)$W2 B1(174)                                  ;INC M
I 'DEC'_'SP'_'IX'B1(65)                                    ;DEC SP,IX
I 'DEC'_'SP'_B1(64)                                        ;DEC SP
I 'DEC'_'SP'B1(64)                                         ;DEC SP
I 'DEC'_'IX'B1(11)                                         ;DEC IX
I 'DEC'_(AB)B1(112)B1(V)                                   ;DEC A/B
I 'DEC'_'(IX)+'B1(152)E(8.)B2(V)                           ;DEC (IX)+N
I 'DEC'_E(14.)$W2 B1(172)                                  ;DEC M


SHIFT INSTRUCTIONS:
-------------------

MNEMONICS:
MENU SHIFT1 = ('CLR'=117, 'CPL'=103, 'NEG'=100, 'RLC'=111, 'RRC'=106)
MENU SHIFT2 = ('SLC'=110, 'ASL'=110, 'SRC'=104, 'LSR'=104, 'ASR'=107, 'TEST'=115)

I (SHIFT1)B1(V)_(AB)B1(V)                                  ;SHIFT1 A/B
I (SHIFT1)B1(V)_'(IX)+'B1(40)E(8.)B2(V)                    ;SHIFT1 (IX)+N
I (SHIFT1)B1(V)_E(14.)$W2 B1(60)                           ;SHIFT1 M
I (SHIFT2)B1(V)_(AB)B1(V)                                  ;SHIFT2 A/B
I (SHIFT2)B1(V)_'(IX)+'B1(40)E(8.)B2(V)                    ;SHIFT2 (IX)+N
I (SHIFT2)B1(V)_E(14.)$W2 B1(60)                           ;SHIFT2 M


STACK INSTRUCTIONS:
-------------------

MNEMONICS:
MENU PP = ('PUSH'=66, 'POP'=62)

REGISTERS:
MENU PPAB = ('A'=0, 'B'=1)

  (PP)B1(V)_(PPAB)B1(V)                                    ;PUSH/POP A/B


CARRY AND OVERFLOW INSTRUCTIONS:
--------------------------------

MNEMONICS:
MENU CV =  ('SETC'=15, 'CLRC'=14, 'SETV'=13, 'CLRV'=12)

  (CV) B1(V)


JUMP INSTRUCTIONS:
------------------

CONDITION CODES:
 T1 =('CS'=45, 'CC'=44, 'MI'=53, 'SS'=53, 'PL'=52, 'SC'=52, 'EQ'=47, 'ZS'=47, 'NE'=
```

```
M T2 =('ZC'=46, 'VS'=51, 'VC'=50, 'LS'=43, 'HI'=42, 'GT'=56, 'GE'=54, 'LE'=57, 'LT'=

FORCING RELATIVE ADDRESSING:
  'JUMP. '_E(14. )$REL B1(40)B2(V1)                    ;JUMP.   .+N'

  'JUMP'_(T1)B1(V)_E(14. )$REL B2(V1)                  ;JUMP,T1 .+N'
  'JUMP'_(T2)B1(V)_E(14. )$REL B2(V1)                  ;JUMP,T2 .+N'
  'JUMP'_'(IX)+'B1(156)E(8. )B2(V)                     ;JUMP (IX)+N
  'JUMP'_E(14. )$REL B1(40)B2(V1)                      ;JUMP .+N'
  'JUMP'_E(14. )$W2 B1(176)                            ;JUMP M


CALL INSTRUCTIONS:
------------------

FORCING RELATIVE ADDRESSING:
  'CALL. '_E(14. )$REL B1(215)B2(V1)                   ;CALL.   .+N'

  'CALL'_'(IX)+'B1(255)E(8. )B2(V)                     ;CALL (IX)+N
  'CALL'_E(14. )$REL B1(215)B2(V1)                     ;CALL .+N'
  'CALL'_E(14. )$W2 B1(275)                            ;CALL M


MISCELLANEOUS INSTRUCTIONS:
---------------------------

MNEMONICS:
MENU DIVERS = ('RET'=71, 'RTI'=73, 'TRAP'=77, 'WAIT'=76, 'ION'=16, 'IOF'=17, 'NOP'

  (DIVERS) B1(V)


                .LIST     L+
```

D. PROGRAMMES DE TRADUCTION ET D'OPTIMISATION DU MOTOROLA 6800

```
1     **
2     **      lecture du fichier input
3     **
4
5     go1     input = syspit                          /f(end)
6
7     **
8     **      recherche de l'instruction VAC correspondant
9     **      a l'instruction lue precedemment
10    **
11
12            **
13            ** instructions ne demandant aucune modification
14            **
15
16            input */8* 'NAME' ** = ''               /s(go2)
17            input */8* 'END' = ''                   /s(go2)
18            input */8* ';*DATA'                     /s(go2)
19
20            **
21            ** instructions VAC demandant de generer plusieurs
22            ** lignes de code CALM
23            **
24
25            input */8* 'MULTI'                      /s(multi)
26            input */8* 'REMVEI'                     /s(remvei)
27            input */8* 'RRESI'                      /s(rresi)
28            input *pr* '    ENTRY'                  /s(entry)
29            input */8* 'SAVE       ' *pr*           /s(save)
30            input */8* 'JLE ' *pr*                  /s(jle)
31            input */8* 'LVPARI     ' *pr*           /s(lvpari)
32            input */8* 'LVPARC     ' *pr*           /s(lvparc)
33            input */8* 'LVCONI     ' *pr*           /s(lvconi)
34            input */8* 'LEI'                        /s(lei)
35            input */8* 'JUMPF      ' *pr*           /s(jumpf)
36            input */8* 'PLUSI'                      /s(plusi)
37            input */8* 'CALL       ' *pr*           /s(call)
38            input */8* 'JUMP       ' *pr*           /s(jump)
39            input */8* 'RETRNI     ' *pr*           /s(retrni)
40            input */8* 'LALOC      ' *pr*           /s(laloc)
41            input */8* 'SVALI'                      /s(svali)
42            input */8* 'SVALC'                      /s(svalc)
43            input */8* 'NEI'                        /s(nei)
44            input */8* 'LVLOCC     ' *pr*           /s(lvlocc)
45            input */8* 'LVPARC     ' *pr*           /s(lvlocc)
46            input */8* 'INDIRC'                     /s(indirc)
47            input */8* 'LAPAR      ' *pr*           /s(laloc)
48            input */8* 'DUPLI'                      /s(dupli)
49            input */8* 'LVLOCI     ' *pr*           /s(lvloci)
50            input */8* 'LAEXT      ' *pr*           /s(laext)
51            input */8* 'ANDI'                       /s(andi)
52            input */8* 'ORI'                        /s(ori)
53            input */8* 'CALLII'                     /s(callii)
54            input */8* 'COMPI'                      /s(compi)
55            input */8* 'DIVI'                       /s(divi)
56            input */8* 'EQI'                        /s(eqi)
57            input */8* 'EXORI'                      /s(exori)
58            input */8* 'INCAFT     ' *pr*           /s(incaft)
59            input */8* 'DECAFT     ' *pr*           /s(decaft)
60            input */8* 'INCBEF     ' *pr*           /s(incbef)
```

```
            input */8* 'INDIRI'                 /s(indiri)
            input */8* 'JUMPI'                  /s(jumpi)
            input */8* 'JUMPT       ' *pr*      /s(jumpt)
            input */8* 'LTI'                    /s(lti)
            input */8* 'GEI'                    /s(gei)
            input */8* 'LSHFTI'                 /s(lshfti)
            input */8* 'RSHFTI'                 /s(rshfti)
            input */8* 'LVEXTI      ' *pr*      /s(lvexti)
            input */8* 'LVEXTC      ' *pr*      /s(lvextc)
            input */8* 'NEGI'                   /s(negi)
            input */8* 'NOTI'                   /s(noti)
            input */8* 'REMI'                   /s(remi)
            input */8* 'SUBTRI'                 /s(subtri)
            input */8* 'SWTCHD'                 /s(swtchd)
            input */8* 'SWTCHH'                 /s(swtchh)
            input */8* 'SWTCHS'                 /s(swtchs)
            input */8* 'GTI'                    /s(gti)
            input */8* 'JGE ' *pr*              /s(jge)
            input */8* 'JLT ' *pr*              /s(jlt)
            input */8* 'JGT ' *pr*              /s(jgt)
            input */8* 'JNE ' *pr*              /s(jne)
            input */8* 'JEQ ' *pr*              /s(jeq)
            input */8* 'DECBEF      ' *pr*      /s(decbef)

    **
    ** lecture d'une instruction VAC du fichier input
    **

go2         syspot = input                      /(go1)




    **
    ** GENERATION DES INSTRUCTIONS CALM CORRESPONDANT
    ** A CHAQUE INSTRUCTION VAC
    **



    ** INSTRUCTION ENTRY
    **
    **      permet de sauver l'environnement a l'entree
    **      d'une procedure
    **

entry       syspot = pr ': LOAD A,LL'
            syspot = 'LOAD B,LH'
            syspot = 'PUSH A'
            syspot = 'PUSH B'           /(go1)

    ** INSTRUCTION GTI
    **
    **      place 1 au sommet de la pile si l'element
    **      au sommet de la pile est > que le suivant
    **      sur la pile, sinon place 0 au sommet de la
    **      pile. Les operandes sont enlevees de la pile
    **
```

```
121     gti     syspot = 'POP B'
122             syspot = 'POP A'
123             syspot = 'INC IX,SP'
124             syspot = 'CALL GTI'     /(go1)
125
126     **
127     ** INSTRUCTION MULTI
128     **
129     **      multiplication entiere des 2 operandes
130     **      au sommet de la pile,le resultat remplacant
131     **      les operandes sur la pile
132     **
133
134
135             **
136             ** appel de la fonction extract.fnc
137             **
138
139     multi   syspot = 'POP B'
140             syspot = 'POP A'
141             syspot = 'INC IX,SP'
142             syspot = 'CALL MULTI'   /(go1)
143
144     **
145     ** INSTRUCTION REMVEI
146     **
147     **      extraction de l'element au sommet de la pile
148     **
149
150     remvei  syspot = 'POP B'
151             syspot = 'POP A'        /(go1)
152
153     **
154     ** INSTRUCTION SAVE
155     **
156     **      calcule la nouvelle valeur du pointeur LL-LH
157     **      qui exprime l'endroit ou se touvent les
158     **      variables locales
159     **
160
161     save    syspot = 'LOAD LH,SP'
162             syspot = 'LOAD B,LH'
163             syspot = 'LOAD A,LL'
164             syspot = 'ADD A,#1'
165             syspot = 'ADDC B,#0'
166             syspot = 'SUB A,#2*' pr
167             syspot = 'SUBC B,#0'
168             syspot = 'LOAD LH,B'
169             syspot = 'LOAD LL,A'
170             syspot = 'LOAD SP,LH'
171             syspot = 'DEC SP'       /(go1)
172
173     **
174     ** INSTRUCTION LVCONI
175     **
176     **      permet de placer une constante entiere
177     **      au sommet de la pile
178     **
179
180     lvconi  syspot = 'LOAD IX,#' pr
```

```
191             syspot = 'LOAD SH,IX'
192             syspot = 'LOAD A,SL'
193             syspot = 'LOAD B,SH'
194             syspot = 'PUSH A'
195             syspot = 'PUSH B'       /(go1)
196
197     **
198     ** INSTRUCTION LEI
199     **
200     **      place 1 au sommet de la pile si l'element
201     **      au sommet de la pile est <= que le suivant
202     **      sur la pile,sinon place 0 au sommet de la
203     **      pile.Les operandes sont enlevees de la pile
204     **
205
206     lei     syspot = 'POP B'
207             syspot = 'POP A'
208             syspot = 'INC IX,SP'
209             syspot = 'CALL LEI'     /(go1)
210
211     **
212     ** INSTRUCTION LVPARI
213     **
214     **      place la valeur d'un prametre sur la pile
215     **
216
217     lvpri   syspot = 'LOAD IX,LH'
218             syspot = 'LOAD B,(IX)+2*' pr
219             syspot = 'LOAD A,(IX)+1+2*' pr
220             syspot = 'PUSH A'
221             syspot = 'PUSH B'       /(go1)
222
223     **
224     ** INSTRUCTION LVPARC
225     **
226     **      place la valeur d'un prametre sur la pile
227     **      (valeur caractere)
228     **
229
230     lvprc   syspot = 'LOAD IX,LH'
231             syspot = 'LOAD A,(IX)+1+2*' pr
232             syspot = 'CLR B'
233             syspot = 'PUSH A'
234             syspot = 'PUSH B'       /(go1)
235
236     **
237     ** INSTRUCTION LVLOCC
238     **
239     **      place la valeur d'une variable locale
240     **      de type caractere sur la pile
241     **
242
243     lvlocc  syspot = 'LOAD IX,LH'
244             syspot = 'LOAD A,(IX)+1+2*' pr
245             syspot = 'CLR B'
246             syspot = 'PUSH A'
247             syspot = 'PUSH B'       /(go1)
248
249     **
250     ** INSTRUCTION INDIRC
```

```
241     **
242     **      l'entier au sommet de la pile est interprete
243     **      comme une adresse. Cette adresse est remplacee
244     **      pr la valeur ou elle pointe(caractere)
245     **
246
247     indirc  syspot = 'POP B'
248             syspot = 'POP A'
249             syspot = 'CALL INDIRC'
250             syspot = 'PUSH A'
251             syspot = 'PUSH B'       /(go1)
252
253     **
254     ** INSTRUCTION PLUSI
255     **
256     **      addition des 2 valeurs au sommet de la pile
257     **
258
259     plusi   syspot = 'POP B'
260             syspot = 'POP A'
261             syspot = 'INC IX,SP'
262             syspot = 'ADD A,(IX)+1'
263             syspot = 'ADDC B,(IX)+0'
264             syspot = 'LOAD (IX)+1,A'
265             syspot = 'LOAD (IX)+0,B'        /(go1)
266
267     **
268     ** INSTRUCTION CALL
269     **
270     **      appel d'une fonction
271     **
272
273     call    syspot = 'CALL ' pr             /(go1)
274
275     **
276     ** INSTRUCTION JUMP
277     **
278     **      branchement inconditionnel
279     **
280
281     jump    syspot = 'JUMP ' pr             /(go1)
282
283     **
284     ** INSTRUCTION RETRNI
285     **
286     **      restitue l'environement a la fin d'une
287     **      procedure
288     **
289
290     retrni  syspot = 'LOAD IX,LH'
291             syspot = 'LOAD IX,(IX)+2*' pr
292             syspot = 'LOAD SSH,IX'
293             syspot = 'LOAD IX,LH'
294             syspot = 'LOAD IX,(IX)+2+2*' pr
295             syspot = 'LOAD SSSH,IX'
296             syspot = 'LOAD 1X,LH'
297             syspot = 'LOAD IX,(IX)+4+2*' pr
298             syspot = 'LOAD A,#' pr
299             syspot = 'JUMP RETRNI'  /(go1)
300
```

```
301     **
302     ** INSTRUCTION LALOC
303     **
304     **      place une adresse locale sur la pile
305     **
306
307     laloc   syspot = 'LOAD A,LL'
308             syspot = 'LOAD B,LH'
309             syspot = 'LOAD IX,#2*' pr
310             syspot = 'LOAD SH,IX'
311             syspot = 'ADD A,SL'
312             syspot = 'ADDC B,SH'
313             syspot = 'PUSH A'
314             syspot = 'PUSH B'       /(go1)
315
316     **
317     ** INSTRUCTION SVALI
318     **
319     **      l'adresse et la valeur d'une variable
320     **      se trouvent au sommet de la pile.Le
321     **      but de l'instruction est d'assigner
322     **      la valeur(sur la pile) a l'adresse
323     **      (sur la pile)
324     **
325
326     svali   syspot = 'POP B'
327             syspot = 'POP A'
328             syspot = 'INC IX,SP'
329             syspot = 'INC SP'
330             syspot = 'INC SP'
331             syspot = 'LOAD IX,(IX)+0'
332             syspot = 'LOAD (IX)+1,A'
333             syspot = 'LOAD (IX)+0,B'
334             syspot = 'PUSH A'
335             syspot = 'PUSH B'       /(go1)
336
337     **
338     ** INSTRUCTION SVALC
339     **
340     **      l'adresse et la valeur d'une variable
341     **      se trouvent au sommet de la pile.Le
342     **      but de l'instruction est d'assigner
343     **      la valeur(sur la pile) a l'adresse
344     **      (sur la pile)
345     **      (la valeur est de type caractere)
346     **
347
348     svalc   syspot = 'POP B'
349             syspot = 'POP A'
350             syspot = 'INC IX,SP'
351             syspot = 'INC SP'
352             syspot = 'INC SP'
353             syspot = 'LOAD IX,(IX)+0'
354             syspot = 'LOAD (IX)+0,A'
355             syspot = 'PUSH A'
356             syspot = 'PUSH B'       /(go1)
357
358     **
359     ** INSTRUCTION NEI
360     **
```

```
361     **      place 1 au sommet de la pile si l'element
362     **      au sommet de la pile est != que le suivant
363     **      sur la pile, sinon place 0 au sommet de la
364     **      pile. Les operandes sont enlevees de la pile
365     **
366
367     nei     syspot = 'POP B'
368             syspot = 'POP A'
369             syspot = 'INC IX,SP'
370             syspot = 'CALL NEI'             /(go1)
371
372     **
373     ** INSTRUCTION LVLOCI
374     **
375     **      place une valeur entiere sur la pile
376     **
377
378     lvloci  syspot = 'LOAD IX,LH'
379             syspot = 'LOAD B,(IX)+2*' pr
380             syspot = 'LOAD A,(IX)+1+2*' pr
381             syspot = 'PUSH A'
382             syspot = 'PUSH B'       /(go1)
383
384     **
385     ** INSTRUCTION LAEXT
386     **
387     **      place une adresse abolue sur la pile
388     **
389
390     laext   syspot = 'LOAD IX,#' pr
391             syspot = 'LOAD SH,IX'
392             syspot = 'LOAD A,SL'
393             syspot = 'LOAD B,SH'
394             syspot = 'PUSH A'
395             syspot = 'PUSH B'       /(go1)
396
397     **
398     ** INSTRUCTION JUMPF
399     **
400     **      branchement conditionnel si l'element au
401     **      sommet de la pile est nul
402     **
403
404     jumpf   syspot = 'POP B'
405             syspot = 'COMP B,#0'
406             syspot = 'JUMP,NE .+10'
407             syspot = 'POP A'
408             syspot = 'COMP A,#0'
409             syspot = 'JUMP,NE .+5'
410             syspot = 'JUMP ' pr     /(go1)
411
412     **
413     ** INSTRUCTION RRESI
414     **
415     **      reserve une place sur la pile pour
416     **      placer le resultat d'une procedure
417     **
418
419     rresi   syspot = 'PUSH B'
420             syspot = 'PUSH B'       /(go1)
```

```
421
422     **
423     ** INSTRUCTION JGT
424     **
425     **      branchement conditionnel en fonction de
426     **      la valeur des 2 operandes au sommet de la pile
427     **      correspond a la seqence:
428     **              GTI
429     **              JUMPT
430     **      ou
431     **              LEI
432     **              JUMPF
433     **
434
435     jgt     syspot = 'POP B'
436             syspot = 'POP A'
437             syspot = 'INC IX,SP'
438             syspot = 'INC SP'
439             syspot = 'INC SP'
440             syspot = 'COMP B,(IX)+0'
441             syspot = 'JUMP,LT .+8'
442             syspot = 'JUMP,GT .+9'
443             syspot = 'COMP A,(IX)+1'
444             syspot = 'JUMP,GE .+5'
445             syspot = 'JUMP ' pr     /(go1)
446
447     **
448     ** INSTRUCTION ORI
449     **
450     **      operation or entre les 2 elements au sommet de la
451     **      pile
452     **
453
454     ori     syspot = 'POP B'
455             syspot = 'POP A'
456             syspot = 'INC IX,SP'
457             syspot = 'OR A,(IX)+1'
458             syspot = 'OR B,(IX)+0'
459             syspot = 'LOAD (IX)+1,A'
460             syspot = 'LOAD (IX)+0,B'        /(go1)
461
462     **
463     ** INSTRUCTION CALLII
464     **
465     **      appel de la fonction dont l'adresse se
466     **      trouve au sommet de la pile
467     **
468
469     callii  syspot = 'LOAD SH,SP'
470             syspot = 'LOAD IX,SH'
471             syspot = 'INC SP'
472             syspot = 'INC SP'
473             syspot = 'CALL (IX)+1'  /(go1)
474
475     **
476     ** INSTRUCTION COMPI
477     **
478     **      complement de la valeur sur la pile
479     **
480
```

```
481     compi   syspot = 'POP B'
482             syspot = 'POP A'
483             syspot = 'CPL A'
484             syspot = 'CPL B'
485             syspot = 'PUSH A'
486             syspot = 'PUSH B'       /(go1)
487
488     **
489     ** INSTRUCTION DUPLI
490     **
491     **      duplicie la valeur au sommet de la pile
492     **
493
494     dupli   syspot = 'POP B'
495             syspot = 'POP A'
496             syspot = 'PUSH A'
497             syspot = 'PUSH B'
498             syspot = 'PUSH A'
499             syspot = 'PUSH B'       /(go1)
500
501     **
502     ** INSTRUCTION EQI
503     **
504     **      place 1 au sommet de la pile si l'element
505     **      au sommet de la pile est = que le suivant
506     **      sur la pile, sinon place 0 au sommet de la
507     **      pile. Les operandes sont enlevees de la pile
508     **
509
510     eqi     syspot = 'POP B'
511             syspot = 'POP A'
512             syspot = 'INC IX,SP'
513             syspot = 'CALL EQI'             /(go1)
514
515     **
516     ** INSTRUCTION EXORI
517     **
518     **      operation ^ entre les 2 operandes au sommet
519     **      de la pile
520     **
521
522     exori   syspot = 'POP B'
523             syspot = 'POP A'
524             syspot = 'INC IX,SP'
525             syspot = 'XOR A,(IX)+1'
526             syspot = 'XOR B,(IX)+0'
527             syspot = 'LOAD (IX)+1,A'
528             syspot = 'LOAD (IX)+0,B'        /(go1)
529
530     **
531     ** INSTRUCTION ANDI
532     **
533     **      operation ^ entre les 2 operandes au sommet
534     **      de la pile
535     **
536
537     andi    syspot = 'POP B'
538             syspot = 'POP A'
539             syspot = 'INC IX,SP'
540             syspot = 'AND A,(IX)+1'
```

```
541             syspot = 'AND B,(IX)+0'
542             syspot = 'LOAD (IX)+1,A'
543             syspot = 'LOAD (IX)+0,B'        /(go1)
544
545     **
546     ** INSTRUCTION INCAFT
547     **
548     **      traduction de "increment after" de C
549     **
550
551     incaft  syspot = 'POP B'
552             syspot = 'POP A'
553             syspot = 'LOAD SH,B'
554             syspot = 'LOAD B,#' pr
555             syspot = 'CALL INCAFT'
556             syspot = 'PUSH A'
557             syspot = 'PUSH B'       /(go1)
558
559     **
560     ** INSTRUCTION DECAFT
561     **
562     **      traduction de "decrement after" de C
563     **
564
565     decaft  syspot = 'POP B'
566             syspot = 'POP A'
567             syspot = 'LOAD SH,B'
568             syspot = 'LOAD B,#' pr
569             syspot = 'CALL DECAFT'
570             syspot = 'PUSH A'
571             syspot = 'PUSH B'       /(go1)
572
573     **
574     ** INSTRUCTION DECBEF
575     **
576     **      traduction de "decrement before" de C
577     **
578
579     decbef  syspot = 'POP B'
580             syspot = 'POP A'
581             syspot = 'LOAD SH,B'
582             syspot = 'LOAD B,#' pr
583             syspot = 'CALL DECBEF'
584             syspot = 'PUSH A'
585             syspot = 'PUSH B'       /(go1)
586
587     **
588     ** INSTRUCTION INCBEF
589     **
590     **      traduction de "increment before" de C
591     **
592
593     incbef  syspot = 'POP B'
594             syspot = 'POP A'
595             syspot = 'LOAD SH,B'
596             syspot = 'LOAD B,#' pr
597             syspot = 'CALL INCBEF'
598             syspot = 'PUSH A'
599             syspot = 'PUSH B'       /(go1)
600
```

```
601     **
602     ** INSTRUCTION INDIRI
603     **
604     **      remplace l'adresse au sommet de la pile
605     **      pr la valeur entiere ou elle pointe
606     **
607
608     indiri  syspot = 'POP B'
609             syspot = 'POP A'
610             syspot = 'CALL INDIRI'
611             syspot = 'PUSH A'
612             syspot = 'PUSH B'       /(go1)
613
614     **
615     ** INSTRUCTION JUMPI
616     **
617     **      branchement a l'adresse au sommet de la pile
618     **
619
620     jumpi   syspot = 'LOAD SH,SP'
621             syspot = 'LOAD IX,SH'
622             syspot = 'INC SP'
623             syspot = 'INC SP'
624             syspot = 'JUMP (IX)+1'  /(go1)
625
626     **
627     ** INSTRUCTION JUMPT
628     **
629     **      branchement conditionnel si la valeur sur
630     **      la pile est != 0
631     **
632
633     jumpt   syspot = 'POP B'
634             syspot = 'COMP B,#0'
635             syspot = 'JUMP,NE .+7'
636             syspot = 'POP A'
637             syspot = 'COMP A,#0'
638             syspot = 'JUMP,EQ .+5'
639             syspot = 'JUMP' pr      /(go1)
640
641     **
642     ** INSTRUCTION LTI
643     **
644     **      place 1 au sommet de la pile si l'element
645     **      au sommet de la pile est < que le suivant
646     **      sur la pile,sinon place 0 au sommet de la
647     **      pile.Les operandes sont enlevees de la pile
648     **
649
650     lti     syspot = 'POP B'
651             syspot = 'POP A'
652             syspot = 'INC IX,SP'
653             syspot = 'CALL LTI'     /(go1)
654
655     **
656     ** INSTRUCTION GEI
657     **
658     **      place 1 au sommet de la pile si l'element
659     **      au sommet de la pile est >= que le suivant
660     **      sur la pile,sinon place 0 au sommet de la
```

```
661     **      pile.Les operandes sont enlevees de la pile
662     **
663
664     gei     syspot = 'POP D'
665             syspot = 'POP A'
666             syspot = 'INC IX,SP'
667             syspot = 'CALL GEI'     /(go1)
668
669     **
670     ** INSTRUCTION LSHFTI
671     **
672     **      instruction << entre les 2 elements
673     **      au sommet de la pile
674     **
675
676     lshfti  syspot = 'POP B'
677             syspot = 'POP A'
678             syspot = 'INC IX,SP'
679             syspot = 'SLC (IX)+1'
680             syspot = 'RLC (IX)+0'
681             syspot = 'DEC A'
682             syspot = 'JUMP,ZC .-5'  /(go1)
683
684     **
685     ** INSTRUCTION RSHFTI
686     **
687     **      instruction >> entre les 2 elements
688     **      au sommet de la pile
689     **
690
691     rshfti  syspot = 'POP B'
692             syspot = 'POP A'
693             syspot = 'INC IX,SP'
694             syspot = 'SRC (IX)+0'
695             syspot = 'RRC (IX)+1'
696             syspot = 'DEC A'
697             syspot = 'JUMP,ZC .-5'  /(go1)
698
699     **
700     ** INSTRUCTION LVEXTI
701     **
702     **      place une valeur sur la pile
703     **      (valeur entiere)
704     **
705
706     lvexti  syspot = 'LOAD IX,' pr
707             syspot = 'LOAD SH,IX'
708             syspot = 'LOAD A,SL'
709             syspot = 'LOAD B,SH'
710             syspot = 'PUSH A'
711             syspot = 'PUSH B'       /(go1)
712
713
714     **
715     ** INSTRUCTION LVEXTC
716     **
717     **      place une valeur sur la pile
718     **      (valeur caractere)
719     **
720
```

```
721     lvextc  syspot = 'LOAD A,' pr
722             syspot = 'CLR B'
723             syspot = 'PUSH A'
724             syspot = 'PUSH B'       /(go1)
725
726     **
727     ** INSTRUCTION NEGI
728     **
729     **      valeur opposee de la valeur sur la pile
730     **
731
732     negi    syspot = 'POP B'
733             syspot = 'POP A'
734             syspot = 'CPL A'
735             syspot = 'CPL B'
736             syspot = 'ADD A,#1'
737             syspot = 'ADDC B,#0'
738             syspot = 'PUSH A'
739             syspot = 'PUSH B'       /(go1)
740
741     **
742     ** INSTRUCTION NOTI
743     **
744     **      operation ! de l'element au sommet de la pile
745     **
746
747     noti    syspot = 'LOAD IX,SP'
748             syspot = 'CLR A'
749             syspot = 'COMP A,(IX)+2'
750             syspot = 'JUMP,NE .+6'
751             syspot = 'COMP A,(IX)+1'
752             syspot = 'JUMP,EQ .+8'
753             syspot = 'CLR (IX)+2'
754             syspot = 'CLR (IX)+1'
755             syspot = 'JUMP .+4'
756             syspot = 'INC (IX)+2'   /(go1)
757
758     **
759     ** INSTRUCTION REMI
760     **
761     **      operation % entre les 2 elements au sommet
762     **      de la pile
763     **
764
765     remi    syspot = 'POP B'
766             syspot = 'POP A'
767             syspot = 'INC IX,SP'
768             syspot = 'CALL DIVREM'
769             syspot = 'INC SP'
770             syspot = 'INC SP'
771             syspot = 'PUSH A'
772             syspot = 'PUSH B'       /(go1)
773
774
775     **
776     ** INSTRUCTION DIVI
777     **
778     **      operation ! entre les 2 elements au sommet
779     **      de la pile
780     **
```

```
781
782     divi    syspot = 'POP B'
783             syspot = 'POP A'
784             syspot = 'INC IX,SP'
785             syspot = 'CALL DIVI'
786             syspot = 'INC SP'
787             syspot = 'INC SP'
788             syspot = 'PUSH A'
789             syspot = 'PUSH B'       /(go1)
790
791     **
792     ** INSTRUCTION SUBTRI
793     **
794     **      operation - entre les 2 elements au sommet
795     **      de la pile
796     **
797
798     subtri  syspot = 'INC IX,SP'
799             syspot = 'INC SP'
800             syspot = 'INC SP'
801             syspot = 'POP B'
802             syspot = 'POP A'
803             syspot = 'SUB A,(IX)+1'
804             syspot = 'SUBC B,(IX)+0'
805             syspot = 'PUSH A'
806             syspot = 'PUSH B'       /(go1)
807
808     **
809     ** INSTRUCTIONS SWTCHD
810     **
811     **      traduction de l'instruction switch de C
812     **
813
814     swtchd  syspot = 'POP B'
815             syspot = 'POP A'
816             syspot = 'JUMP SWTCHD'  /(go1)
817
818     **
819     ** INSTRUCTIONS SWTCHH
820     **
821     **      traduction de l'instruction switch de C
822     **
823
824     swtchh  syspot = 'POP B'
825             syspot = 'POP A'
826             syspot = 'JUMP SWTCHH'  /(go1)
827
828     **
829     ** INSTRUCTIONS SWTCHS
830     **
831     **      traduction de l'instruction switch de C
832     **
833
834     swtchs  syspot = 'POP B'
835             syspot = 'POP A'
836             syspot = 'JUMP SWTCHS'  /(go1)
837
838     **
839     ** INSTRUCTION JEQ
840     **
```

```
841     **      branchement conditionnel en fonction de
842     **      la valeur des 2 operandes au sommet de la pile
843     **      correspond a la seqence:
844     **              EQI
845     **              JUMPT
846     **      ou
847     **              NEI
848     **              JUMPF
849     **
850
851     jeq     syspot = 'POP B'
852             syspot = 'POP A'
853             syspot = 'LOAD SH,A'
854             syspot = 'POP A'
855             syspot = 'COMP A,B'
856             syspot = 'JUMP,NE .+10'
857             syspot = 'POP B'
858             syspot = 'COMP B,SH'
859             syspot = 'JUMP,NE .+5'
860             syspot = 'JUMP ' pr     /(go1)
861
862     **
863     ** INSTRUCTION JNE
864     **
865     **      branchement conditionnel en fonction de
866     **      la valeur des 2 operandes au sommet de la pile
867     **      correspond a la seqence:
868     **              NEI
869     **              JUMPT
870     **      ou
871     **              EQI
872     **              JUMPF
873     **
874
875     jne     syspot = 'POP B'
876             syspot = 'POP A'
877             syspot = 'LOAD SH,A'
878             syspot = 'POP A'
879             syspot = 'COMP A,B'
880             syspot = 'JUMP,NE .+7'
881             syspot = 'POP A'
882             syspot = 'COMP A,SH'
883             syspot = 'JUMP,EQ .+5'
884             syspot = 'JUMP ' pr     /(go1)
885
886     **
887     ** INSTRUCTION JGE
888     **
889     **      branchement conditionnel en fonction de
890     **      la valeur des 2 operandes au sommet de la pile
891     **      correspond a la seqence:
892     **              GEI
893     **              JUMPT
894     **      ou
895     **              LTI
896     **              JUMPF
897     **
898
899     jge     syspot = 'POP B'
900             syspot = 'POP A'
```

```
901             syspot = 'INC IX,SP'
902             syspot = 'INC SP'
903             syspot = 'INC SP'
904             syspot = 'COMP B,(IX)+0'
905             syspot = 'JUMP,LT .+8'
906             syspot = 'JUMP,GT .+9'
907             syspot = 'COMP A,(IX)+1'
908             syspot = 'JUMP,GT .+5'
909             syspot = 'JUMP ' pr      /(go1)
910
911     **
912     ** INSTRUCTION JLE
913     **
914     **      branchement conditionnel en fonction de
915     **      la valeur des 2 operandes au sommet de la pile
916     **      correspond a la seqence:
917     **              LEI
918     **              JUMPT
919     **      ou
920     **              GTI
921     **              JUMPF
922     **
923
924     jle     syspot = 'POP B'
925             syspot = 'POP A'
926             syspot = 'INC IX,SP'
927             syspot = 'INC SP'
928             syspot = 'INC SP'
929             syspot = 'COMP B,(IX)+0'
930             syspot = 'JUMP,GT .+8'
931             syspot = 'JUMP,LT .+9'
932             syspot = 'COMP A,(IX)+1'
933             syspot = 'JUMP,LT .+5'
934             syspot = 'JUMP ' pr      /(go1)
935
936     **
937     ** INSTRUCTION JLT
938     **
939     **      branchement conditionnel en fonction de
940     **      la valeur des 2 operandes au sommet de la pile
941     **      correspond a la seqence:
942     **              LTI
943     **              JUMPT
944     **      ou
945     **              GEI
946     **              JUMPF
947     **
948
949     jlt     syspot = 'POP B'
950             syspot = 'POP A'
951             syspot = 'INC IX,SP'
952             syspot = 'INC SP'
953             syspot = 'INC SP'
954             syspot = 'COMP B,(IX)+0'
955             syspot = 'JUMP,GT .+8'
956             syspot = 'JUMP,LT .+9'
957             syspot = 'COMP A,(IX)+1'
958             syspot = 'JUMP,LE .+5'
959             syspot = 'JUMP ' pr      /(go1)
960
```

```
961     end     syspot = ''
```

```
1
2       ** INITIALISATIONS
3       **
4       **      ces variables,initialisees a 0 seront
5       **      mises a 1 si au moins une fois la
6       **      routine dont elles portent le nom
7       **      se retrouve dans le texte input
8       **
9
10              retrni  =       '0'
11              indirc  =       '0'
12              indiri  =       '0'
13              incbef  =       '0'
14              decbef  =       '0'
15              incaft  =       '0'
16              decaft  =       '0'
17              divi    =       '0'
18              remi    =       '0'
19              divrem  =       '0'
20              multi   =       '0'
21              eqi     =       '0'
22              nei     =       '0'
23              lti     =       '0'
24              lei     =       '0'
25              gti     =       '0'
26              gei     =       '0'
27              swtchs  =       '0'
28
29      **
30      ** SEQUENCE D'INITIALISATION DU PROGRAMME
31      **
32      **      definition des variables de travail
33      **      initialisation des pointeurs et de l'
34      **      adresse debut du programme objet et de
35      **      la pile en memoire
36      **
37
38              syspot  =       '.RDX 10.'
39              syspot  =       'LH: .BYTE 0'
40              syspot  =       'LL: .BYTE 0'
41              syspot  =       'SH: .BYTE 0'
42              syspot  =       'SL: .BYTE 0'
43              syspot  =       'SSH: .BYTE 0'
44              syspot  =       'SSL: .BYTE 0'
45              syspot  =       'SSSH: .BYTE 0'
46              syspot  =       'SSSL: .BYTE 0'
47              syspot  =       'SSSSH: .BYTE 0'
48              syspot  =       'SSSSL: .BYTE 0'
49              syspot  =       'ADRINIT: .WORD 1000'
50              syspot  =       'DEBUT: .LOC DEBUT'
51              syspot  =       'LOAD SP,ADRINIT'
52              syspot  =       'LOAD IX,ADRINIT'
53              syspot =        'INC IX'
54              syspot  =       'LOAD LH,IX'
55              syspot  =       'CLR A'
56              syspot  =       'PUSH A'
57              syspot  =       'PUSH A'
58              syspot  =       'PUSH A'
59              syspot  =       'PUSH A'
60              syspot  =       'CALL MAIN'
```

```
61              syspot  =       'FIN: WAIT'
62
63      **
64      ** LECTURE DU FICHIER INPUT
65      **
66
67      read1   input1  =       syspit  /f(fin2)
68              input2  =       syspit  /f(fin3)
69              input3  =       syspit  /f(fin4)
70      read4   input4  =       syspit  /f(fin1)
71
72      **
73      ** RECHERCHE DE SEQUENCES D'INSTRUCTIONS CONTRACTABLES
74      **
75
76
77              **
78              ** premiere sequence
79              **
80
81      test1   input1 ** 'LOAD (IX)+1,A'       /f(test2)
82              input2 ** 'LOAD (IX)+0,B'       /f(t2)
83              input3 ** 'POP B'               /f(test2)
84              input4 ** 'POP A'               /f(test2)
85              syspot = 'INC SP'
86              syspot = 'INC SP'               /(read1)
87
88              **
89              ** deuxieme sequence
90              **
91
92      test2   input1 ** 'PUSH A'      /f(chang)
93              input2 ** 'PUSH B'      /f(chang)
94              input3 ** 'POP B'       /f(chang)
95              input4 ** 'POP A'       /s(read1)
96
97
98      **
99      ** test si l'instruction definie par input1
100     ** correspond a une fonction predefinie
101     **
102
103     chang   ap.f(input1)
104
105     **
106     ** LECTURE ET INTERCHANGEMENT
107     **
108
109             syspot  =       input1
110             input1  =       input2
111             input2  =       input3
112             input3  =       input4  /(read4)
113
114     **
115     ** SEQUENCE DE FIN DU PROGRAMME
116     **
117     **      verification si les dernieres instuctions
118     **      correspondent a des appels de fonctions
119     **      predefinies
120     **
```

```
121
122     fin1    ap.f(input1)
123             syspot = input1
124     fin2    ap.f(input2)
125             syspot = input2
126     fin3    ap.f(input3)
127             syspot = input3
128     fin4    ap.f(input4)
129             syspot = input4
130             t.f(par)
131
132     **
133     ** GENERATION DU CODE DES ROUTINES RENCONTREES
134     **
135     **      ceci est realise par la fonction test.fnc
136     **
137
138             test.fnc(par)
139
140
141
142             syspot = '.END'
143
144
145
146
147
148
149
150
151
152     **
153     ** FONCTION     appel.fnc
154     **
155     **
156     **
157     **      le role de cette fonction est de verifier
158     **      si l'instruction qu'on lui passe comme
159     **      parametre correspond a un appel d'une
160     **      routine predefinie.Si c'est le cas elle
161     **      positionne la variable correspondant
162     **      au nom de la routine a 1
163     **
164     **
165     **
166
167     define  appel.fnc(input)
168     appel2  input ** 'INDIRC'       /f(appel3)
169             indirc = '1'            /(return)
170     appel3  input ** 'RETRNI'       /f(appel4)
171             retrni = '1'            /(return)
172     appel4  input ** 'INDIRI'       /f(appel5)
173             indiri = '1'            /(return)
174     appel5  input ** 'INCBEF'       /f(appel6)
175             incbef = '1'            /(return)
176     appel6  input ** 'DECBEF'       /f(appel7)
177             decbef = '1'            /(return)
178     appel7  input ** 'INCAFT'       /f(appel8)
179             incaft = '1'            /(return)
180     appel8  input ** 'DECAFT'       /f(appel9)
```

```
181             decaft = '1'                    /(return)
182     appel9  input ** 'DIVI'                 /f(appel10)
183             divi = '1'
184             divrem = '1'                    /(return)
185     appel10 input ** 'MULTI'                /f(appel11)
186             multi = '1'                     /(return)
187     appel11 input ** 'EQI'                  /f(appel12)
188             eqi = '1'                       /(return)
189     appel12 input ** 'NEI'                  /f(appel13)
190             nei = '1'                       /(return)
191     appel13 input ** 'LEI'                  /f(appel14)
192             lei = '1'                       /(return)
193     appel14 input ** 'LTI'                  /f(appel15)
194             lti = '1'                       /(return)
195     appel15 input ** 'GEI'                  /f(appel16)
196             gei = '1'                       /(return)
197     appel16 input ** 'GTI'                  /f(appel17)
198             gti = '1'                       /(return)
199     appel17 input ** 'SWTCHS'               /f(appel18)
200             swtchs = '1'                    /(return)
201     appel18 input ** 'REMI'                 /f(freturn)
202             remi = '1'
203             divrem = '1'                    /(return)
204     define  test.fnc(par)
205
206     **
207     ** FONCTION      test.fnc
208     **
209     **
210     **
211     **      le role de cette fonction est de generer
212     **      le code correspondant aux routines
213     **      rencontrees dans le texte input
214     **
215     **
216     **
217     **
218     **
219
220     test.fnc2       indirc '1'                  /f(test.fnc3)
221             syspot = 'INDIRC:'
222             syspot = 'LOAD SL,A'
223             syspot = 'LOAD SH,B'
224             syspot = 'LOAD IX,SH'
225             syspot = 'LOAD A,(IX)+0'
226             syspot = 'CLR B'
227             syspot = 'RET'
228     test.fnc3       retrni '1'                  /f(test.fnc4)
229             syspot = 'RETRNI:'
230             syspot = 'LOAD SH,IX'
231             syspot = 'SLC SL'
232             syspot = 'SLC A'
233             syspot = 'ADD A,#9'
234             syspot = 'ADD A,SL'
235             syspot = 'CLR B'
236             syspot = 'ADD A,LL'
237             syspot = 'ADDC B,LH'
238             syspot = 'LOAD SL,A'
239             syspot = 'LOAD SH,B'
240             syspot = 'CLR A'
```

```
241             syspot = 'CLR B'
242             syspot = 'LOAD SSSSH,SP'
243             syspot = 'LOAD IX,SSSSH'
244             syspot = 'INC IX'
245             syspot = 'COMP IX,LH'
246             syspot = 'JUMP,EQ .+4'
247             syspot = 'POP A'
248             syspot = 'POP B'
249             syspot = 'LOAD IX,SSH'
250             syspot = 'LOAD LH,IX'
251             syspot = 'LOAD SP,SH'
252             syspot = 'DEC SP'
253             syspot = 'PUSH B'
254             syspot = 'PUSH A'
255             syspot = 'LOAD IX,SSSH'
256             syspot = 'JUMP (IX)+0'
257     test.fnc4       divrem '1'                              /f(test.fnc5)
258             syspot = 'DIVREM:'
259             syspot = 'CLR SSL'
260             syspot = 'CALL TESTNEG'
261             syspot = 'LOAD SL,A'
262             syspot = 'LOAD SH,B'
263             syspot = 'LOAD A,(IX)+1'
264             syspot = 'LOAD B,(IX)+0'
265             syspot = 'CALL TESTNEG'
266             syspot = 'LOAD IX,#0'
267             syspot = 'COMP B,SH'
268             syspot = 'JUMP,LT .+15'
269             syspot = 'JUMP,GT .+6'
270             syspot = 'COMP A,SL'
271             syspot = 'JUMP,LT .+9'
272             syspot = 'SUB A,SL'
273             syspot = 'SUBC B,SH'
274             syspot = 'INC IX'
275             syspot = 'JUMP .-15'
276             syspot = 'RET'
277             syspot = 'TESTNEG:'
278             syspot = 'TEST B'
279             syspot = 'JUMP,GE .+7'
280             syspot = 'INC SSL'
281             syspot = 'CALL POSIT'
282             syspot = 'RET'
283             syspot = 'POSIT:'
284             syspot = 'CPL A'
285             syspot = 'CPL B'
286             syspot = 'ADD A,#1'
287             syspot = 'ADDC B,#0'
288             syspot = 'RET'
289     test.fnc5       indiri '1'                      /f(test.fnc6)
290             syspot = 'INDIRI:'
291             syspot = 'LOAD SL,A'
292             syspot = 'LOAD SH,B'
293             syspot = 'LOAD IX,SH'
294             syspot = 'LOAD A,(IX)+1'
295             syspot = 'LOAD B,(IX)+0'
296             syspot = 'RET'
297     test.fnc6       incbef '1'                      /f(test.fnc7)
298             syspot = 'INCBEF:'
299             syspot = 'LOAD SL,A'
300             syspot = 'LOAD IX,SH'
```

```
301             syspot = 'LOAD A,(IX)+1'
302             syspot = 'ADD A,B'
303             syspot = 'LOAD (IX)+1,A'
304             syspot = 'LOAD B,(IX)+0'
305             syspot = 'ADDC B,#0'
306             syspot = 'LOAD (IX)+0,B'
307             syspot = 'RET'
308     test.fnc7       decbef '1'                      /f(test.fnc8)
309             syspot = 'DECBEF:'
310             syspot = 'LOAD SL,A'
311             syspot = 'LOAD IX,SH'
312             syspot = 'LOAD A,(IX)+1'
313             syspot = 'SUB A,B'
314             syspot = 'LOAD (IX)+1,A'
315             syspot = 'LOAD B,(IX)+0'
316             syspot = 'SUBC B,#0'
317             syspot = 'LOAD (IX)+0,B'
318             syspot = 'RET'
319     test.fnc8       incaft '1'                      /f(test.fnc9)
320             syspot = 'INCAFT:'
321             syspot = 'LOAD SL,A'
322             syspot = 'LOAD IX,SH'
323             syspot = 'LOAD A,(IX)+1'
324             syspot = 'PUSH A'
325             syspot = 'ADD A,B'
326             syspot = 'LOAD (IX)+1,A'
327             syspot = 'LOAD B,(IX)+0'
328             syspot = 'PUSH B'
329             syspot = 'ADDC B,#0'
330             syspot = 'LOAD (IX)+0,B'
331             syspot = 'POP A'
332             syspot = 'POP B'
333             syspot = 'RET'
334     test.fnc9       decaft '1'                      /f(test.fnc10)
335             syspot = 'DECAFT:'
336             syspot = 'LOAD SL,A'
337             syspot = 'LOAD IX,SH'
338             syspot = 'LOAD A,(IX)+1'
339             syspot = 'PUSH A'
340             syspot = 'SUB A,B'
341             syspot = 'LOAD (IX)+1,A'
342             syspot = 'LOAD B,(IX)+0'
343             syspot = 'PUSH B'
344             syspot = 'SUBC B,#0'
345             syspot = 'LOAD (IX)+0,B'
346             syspot = 'POP A'
347             syspot = 'POP B'
348             syspot = 'RET'
349     test.fnc10      eqi '1'                         /f(test.fnc11)
350             syspot = 'EQI:'
351             syspot = 'COMP B,(IX)+0'
352             syspot = 'JUMP,NE .+8'
353             syspot = 'LOAD B,#1'
354             syspot = 'COMP A,(IX)+1'
355             syspot = 'JUMP,EQ .+3'
356             syspot = 'CLR B'
357             syspot = 'CLR A'
358             syspot = 'LOAD (IX)+0,B'
359             syspot = 'LOAD (IX)+1,A'
360             syspot = 'RET'
```

```
test.fnc11      nei '1'                         /f(test.fnc12)
                syspot = 'NEI:'
                syspot = 'COMP B,(IX)+0'
                syspot = 'JUMP,NE .+7'
                syspot = 'CLR B'
                syspot = 'COMP A,(IX)+1'
                syspot = 'JUMP,EQ .+4'
                syspot = 'LOAD B,#1'
                syspot = 'CLR A'
                syspot = 'LOAD (IX)+0,B'
                syspot = 'LOAD (IX)+1,A'
                syspot = 'RET'
test.fnc12      lei '1'                         /f(test.fnc13)
                syspot = 'LEI:'
                syspot = 'COMP B,(IX)+0'
                syspot = 'JUMP,LT .+12'
                syspot = 'JUMP,GT .+6'
                syspot = 'COMP A,(IX)+1'
                syspot = 'JUMP,LT .+6'
                syspot = 'LOAD A,#1'
                syspot = 'JUMP .+3'
                syspot = 'CLR A'
                syspot = 'CLR B'
                syspot = 'LOAD (IX)+1,A'
                syspot = 'LOAD (IX)+0,B'        /(return)
                syspot = 'RET'
test.fnc13      lti '1'                         /f(test.fnc14)
                syspot = 'LTI:'
                syspot = 'COMP B,(IX)+0'
                syspot = 'JUMP,LT .+12'
                syspot = 'JUMP,GT .+6'
                syspot = 'COMP A,(IX)+1'
                syspot = 'JUMP,LE .+6'
                syspot = 'LOAD A,#1'
                syspot = 'JUMP .+3'
                syspot = 'CLR A'
                syspot = 'CLR B'
                syspot = 'LOAD (IX)+1,A'
                syspot = 'LOAD (IX)+0,B'        /(return)
                syspot = 'RET'
test.fnc14      gei '1'                         /f(test.fnc15)
                syspot = 'GEI:'
                syspot = 'COMP B,(IX)+0'
                syspot = 'JUMP,GT .+12'
                syspot = 'JUMP,LT .+6'
                syspot = 'COMP A,(IX)+1'
                syspot = 'JUMP,GT .+6'
                syspot = 'LOAD A,#1'
                syspot = 'JUMP .+3'
                syspot = 'CLR A'
                syspot = 'CLR B'
                syspot = 'LOAD (IX)+1,A'
                syspot = 'LOAD (IX)+0,B'        /(return)
                syspot = 'RET'
test.fnc15      gti '1'                         /f(test.fnc16)
                syspot = 'GTI:'
                syspot = 'COMP B,(IX)+0'
                syspot = 'JUMP,GT .+12'
                syspot = 'JUMP,LT .+6'
                syspot = 'COMP A,(IX)+1'
```

```
421             syspot = 'JUMP,GE .+6'
422             syspot = 'LOAD A,#1'
423             syspot = 'JUMP .+3'
424             syspot = 'CLR A'
425             syspot = 'CLR B'
426             syspot = 'LOAD (IX)+1,A'
427             syspot = 'LOAD (IX)+0,B'        /(return)
428             syspot = 'RET'
429     test.fnc16      multi '1'               /f(test.fnc17)
430             syspot = 'MULTI:'
431             syspot = 'LOAD SL,A'
432             syspot = 'LOAD SH,B'
433             syspot = 'CLR A'
434             syspot = 'CLR B'
435             syspot = 'SRC SH'
436             syspot = 'RRC SL'
437             syspot = 'JUMP,CC .+6'
438             syspot = 'ADD A,(IX)+1'
439             syspot = 'ADDC B,(IX)+0'
440             syspot = 'SLC (IX)+1'
441             syspot = 'RLC (IX)+0'
442             syspot = 'PUSH A'
443             syspot = 'LOAD A,(IX)+0'
444             syspot = 'COMP A,#0'
445             syspot = 'JUMP,NE .+8'
446             syspot = 'LOAD A,(IX)+1'
447             syspot = 'COMP A,#0'
448             syspot = 'JUMP,EQ .+5'
449             syspot = 'POP A'
450             syspot = 'JUMP .-28'
451             syspot = 'LOAD (IX)+1,A'
452             syspot = 'LOAD (IX)+0,B'
453             syspot = 'RET'
454     test.fnc17      swtchs '1'                      /f(test.fnc18)
455             syspot = 'SWTCHS:'
456             syspot = 'LOAD SSSSH,B'
457             syspot = 'LOAD SSSSL,A'
458             syspot = 'POP A'
459             syspot = 'POP B'
460             syspot = 'LOAD SL,A'
461             syspot = 'LOAD SH,B'
462             syspot = 'LOAD IX,SH'
463             syspot = 'LOAD IX,(IX)+2'
464             syspot = 'LOAD SSSH,IX'
465             syspot = 'LOAD IX,SH'
466             syspot = 'LOAD IX,(IX)+0'
467             syspot = 'SWTCHS2:'
468             syspot = 'LOAD SSH,IX'
469             syspot = 'LOAD IX,(IX)+0'
470             syspot = 'COMP IX,SSSSH'
471             syspot = 'JUMP,EQ SWTCHS1'
472             syspot = 'LOAD IX,SSH'
473             syspot = 'ADD A,#2'
474             syspot = 'ADDC B,#0'
475             syspot = 'COMP IX,SSSH'
476             syspot = 'JUMP,GE SWTCHS1'
477             syspot = 'INC IX'
478             syspot = 'INC IX'
479             syspot = 'JUMP SWTCHS2'
480             syspot = 'SWTCHS1:'
```

```
481             syspot = 'LOAD SL,A'
482             syspot = 'LOAD SH,B'
483             syspot = 'LOAD IX,SH'
484             syspot = 'JUMP (IX)+4'
485     test.fnc18      divi    '1'             /f(test.fnc19)
486             syspot = 'DIVI:'
487             syspot = 'CALL DIVREM'
488             syspot = 'LOAD A,SSL'
489             syspot = 'COMP A,#1'
490             syspot = 'JUMP,NE .+11'
491             syspot = 'LOAD SH,IX'
492             syspot = 'LOAD A,SL'
493             syspot = 'LOAD B,SH'
494             syspot = 'CALL POSIT'
495             syspot = 'RET'
496     test.fnc19      remi    '1'             /f(freturn)
497             syspot = 'REMI:'
498             syspot = 'CALL DIVREM'
499             syspot = 'INC SP'
500             syspot = 'INC SP'
501             syspot = 'RET                   /(return)
502     end     syspot = ''
```

ANNEXE 3

ALGORITHMES CALM DU MOTOROLA 6800

ALGORITHMES DES INSTRUCTIONS VAC POUR
=====================================

MOTOROLA 6800
=============

CTION ENTRY

```
LOAD A, LL
LOAD B, LH
PUSH A
PUSH B          ;sauvetage de l'adresse de base locale LH-LL sur la pile
```

CTION GTI

```
POP B
POP A           ;B et A contiennent l'operande 2
INC IX, SP      ;IX pointe sur l'operande 1
CALL GTI

GTI:
COMP B, (IX)+0  ;comparaison parties high
JUMP, GT .+12   ;branchement si >
JUMP, LT .+6    ;branchement si <
COMP A, (IX)+1  ;comparaison parties low
JUMP, GE .+6    ;branchement si >=
LOAD A, #1
JUMP .+3
CLR A
CLR B           ;0 dans A et B
LOAD (IX)+1, A
LOAD (IX)+0, B  ;resultat sur la pile 0 ou 1
RET
```

CTION MULTI

```
POP B
POP A           ;extraire operande 2
INC IX, SP      ;IX pointe sur operande 1
CALL MULTI

MULTI:
LOAD SL, A
LOAD SH, B      ;sauver operande 2 dans SH-SL
CLR A
CLR B           ;0 dans A et B (zone resultat)
SRC SH
RRC SL          ;shift a droite du multiplicateur
JUMP, CC .+6    ;branchement si bit = 0
ADD A, (IX)+1
ADDC B, (IX)+0  ;addition au resultat
SLC (IX)+1
RLC (IX)+0      ;shift a gauche du multiplicande
```

```
    PUSH A              ;sauver A sur la pile
    LOAD A,(IX)+0
    COMP A,#0
    JUMP,NE .+8
    LOAD A,(IX)+1
    COMP A,#0           ;verifier si multiplicande est nul
    JUMP,EQ .+5
    POP A               ;restituer la valeur de A
    JUMP .-28           ;iteration
    LOAD (IX)+1,A
    LOAD (IX)+0,B       ;sauver le resultat sur la pile
    RET

RUCTION REMVEI
--------------
    POP B
    POP A               ;enlever l'element aus sommet de la pile


RUCTION SAVE
------------
    LOAD LH,SP          ;(SP) dans LH-LL(adresse de base locale
    LOAD B,LH
    LOAD A,LL           ;LH-LL dans accumulateurs A et B
    ADD A,#1
    ADDC B,#0
    SUB A,#2* par
    SUBC B,#0           ;calcul de la nouvelle adresse de base locale
    LOAD LH,B
    LOAD LL,A           ;adresse de base locale dans LH-LL
    LOAD SP,LH
    DEC SP              ;calcul de la nouvelle valeur du SP


UCTION LVCONI
-------------
    LOAD IX,# par       ;place la constante dans IX
    LOAD SH,IX
    LOAD A,SL
    LOAD B,SH           ;place la constante dans A et B
    PUSH A
    PUSH B              ;sauvetage sur la pile


JCTION LEI
----------
    POP B
    POP A               ;extraction operande 2
    INC IX,SP           ;IX pointe sur operande 1
    CALL LEI

    LEI:
    COMP B,(IX)+0       ;comparaison parties high
    JUMP,LT .+12        ;branchement si <
    JUMP,GT .+6         ;branchement si >
    COMP A,(IX)+1       ;comparaison parties low
    JUMP,LT .+6         ;branchement si <
```

```
  LOAD A,#1
  JUMP .+3
  CLR A
  CLR B
  LOAD (IX)+1,A    ;sauvetage sur la pile
  LOAD (IX)+0,B    ;sauvetage sur la pile
  RET


CTION LVPARI
------------
  LOAD IX,LH       ;place l'adresse de base locale dans IX
  LOAD B,(IX)+2*pr
  LOAD A,(IX)+1+2*pr
                   ;place la valeur du parametre dans A et B
  PUSH A
  PUSH B           ;sauvetage sur la pile


CTION LVPARC
------------
  LOAD IX,LH       ;place l'adresse de base locale dans IX
  LOAD A,(IX)+1+2*pr
                   ;place la valeur du parametre dans A
  CLR B            ;B=0
  PUSH A
  PUSH B           ;sauvetage sur la pile


CTION LVLOCC
------------
  LOAD IX,LH       ;place l'adresse de base locale dans IX
  LOAD A,(IX)+2* pr        ;place la valeur caractere dans A
  CLR B
  PUSH A
  PUSH B           ;sauvetage sur la pile


CTION INDIRC
------------
  POP B
  POP A            ;extraction adresse
  CALL INDIRC
  PUSH A
  PUSH B           ;sauver le resultat sur la pile

  INDIRC:
  LOAD SL,A
  LOAD SH,B
  LOAD IX,SH       ;charger l'adresse dans IX
  LOAD A,(IX)+0    ;mettre la valeur ou pointe IX dans A
  CLR B
  RET


 TION PLUSI
```

```
----------
  POP B
  POP A              ;extraire operande 2
  INC IX,SP          ;IX pointe sur operande 1
  ADD A,(IX)+1
  ADDC B,(IX)+0      ;addition
  LOAD (IX)+1,A
  LOAD (IX)+0,B      ;sauver le resultat sur la pile


JCTION CALL
----------
  CALL  pr           ;appel de la fonction


JCTION JUMP
----------
  JUMP  pr           ;branchement a par


CTION RETRNI
------------
  LOAD IX,LH         ;charge l'adresse de base locale dans IX
  LOAD IX,(IX)+2* pr       ;charge l'ancienne adresse de base locale dans I
  LOAD SSH,IX        ;sauve cette adresse dans SSH
  LOAD IX,LH
  LOAD IX,(IX)+2+2* pr     ;charge l'adresse de retour dans IX
  LOAD SSH,IX        ;sauve cette adresse dans SSSH
  LOAD IX,LH
  LOAD IX,(IX)+4+2* pr     ;charge le nombre de parametre dans IX
  LOAD A,# pr        ;charge le nombre de variables locales dans A
  JUMP RETRNI

  RETRNI:
  LOAD SH,IX         ;sauve le nombre de parametre dans SH et SL
  SLC SL             ;nombre de parametre * 2    SH=0
  SLC A              ;nombre de variables locales * 2
  ADD A,#8
  ADD A,SL
  CLR B
  ADD A,LL
  ADDC B,LH          ;calcul du deplacement
  LOAD SL,A
  LOAD SH,B          ;sauve le deplacement dans SH et SL
  CLR A
  CLR B              ;A et B =0
  LOAD SSSSH,SP
  LOAD IX,SSSSH      ;charge le stack pointer dans IX
  INC IX             ;ajuste IX
  COMP IX,LH         ;la routine a-t-elle un resultat sur la pile
  JUMP,EQ .+4        ;branche si non
  POP A
  POP B              ;extrais le resultat
  LOAD IX,SSH
  LOAD LH,IX         ;LH et LL contienne l'adresse de base locale
                     ;de la fonction appelante
  LOAD SP,SH
```

```
  DEC SH            ;calcul du nouveau stack pointer
  PUSH B
  PUSH A            ;place le resultat de la fonction sur la pile
  LOAD IX,SSSH
  JUMP (IX)+0       ;branche a l'adresse de retour


CTION LALOC
-----------
  LOAD A,LL
  LOAD B,LH         ;charge l'adresse de base locale dans A et B
  LOAD IX,#2* pr    ;charge le deplacement dans IX
  LOAD SH,IX
  ADD A,SL
  ADDC B,SH         ;calcul de l'adresse
  PUSH A
  PUSH B            ;sauvetage sur la pile


CTION SVALI
-----------
  POP B
  POP A             ;extrais la valeur entiere dans A et B
  INC IX,SP         ;IX pointe sur l'adresse au sommet de la pile
  INC SP
  INC SP            ;ajustement du stack pointer
  LOAD IX,(IX)+0    ;charge l'adresse au sommet de la pile dans IX
  LOAD (IX)+1,A
  LOAD (IX)+0,B     ;sauve la valeur a l'adresse voulue
  PUSH A
  PUSH B            ;sauvetage sur la pile


CTION SVALC
-----------
  POP B
  POP A             ;extrais la valeur caractere dans A (B=0)
  INC IX,SP         ;IX pointe sur l'adresse au sommet de la pile
  INC SP
  INC SP            ;ajuste le stack pointer
  LOAD IX,(IX)+0    ;charge l'adresse au sommet de la pile dans IX
  LOAD (IX)+0,A     ;sauve la valeur a l'adresse voulue
  PUSH A
  PUSH B            ;sauvetage sur la pile


CTION NEI
---------
  POP B
  POP A             ;extrais l'operande 2 du sommet de la pile
  INC IX,SP         ;IX pointe sur l'operande 1
  CALL NEI

  NEI:
  COMP B,(IX)+0     ;comparaison parties high des 2 operandes
  JUMP,NE .+7       ;branche si !=
  CLR B             ;B=0
```

```
  JUMP,EQ .+4        ;branche si ==
  LOAD B,#1          ;B=1
  CLR A              ;A=0
  LOAD (IX)+1,A
  LOAD (IX)+0,B      ;sauvetage sur la pile
  RET


CTION LVLOCI
------------
  LOAD IX,LH         ;charge l'adresse de base locale dans IX
  LOAD B,(IX)+2* pr
  LOAD A,(IX)+1+2* pr
                     ;charge la valeur entiere dans A et B
  PUSH A
  PUSH B             ;sauvetage sur la pile


CTION LAEXT
-----------
  LOAD IX,# pr       ;charge l'adresse dans IX
  LOAD SH,IX
  LOAD A,SL
  LOAD B,SH          ;charge l'adresse dans A et B
  PUSH A
  PUSH B             ;sauvetage sur la pile


CTION JUMPF
-----------
  POP A              ;extrais partie high de l'operande
  COMP A,#0
  JUMP,NE .+9        ;branche si A !=0
  POP B              ;extrais partie low de l'operande
  COMP B,#0
  JUMP,NE .+5        ;branche si B!=0
  JUMP  pr


CTION RRESI
-----------
  PUSH B
  PUSH B             ;reserve un emplacement sur la pile


CTION JLE
---------
  POP B
  POP A              ;extrais l'operande 2
  INC IX,SP          ;IX pointe sur l'operande 1
  INC SP
  INC SP             ;ajuste le stack pointer
  COMP B,(IX)+0      ;compare les parties high des operandes
  JUMP,GT .+8        ;branche si >
  JUMP,LT .+9        ;branche si <
```

```
  COMP A,(IX)+1     ;compare les parties low des operandes
  JUMP,LT .+5       ;branche si <
  JUMP   pr


CTION ORI
---------
  POP B
  POP A             ;extrais l'operande 2
  INC IX,SP         ;IX pointe sur l'operande 1
  OR A,(IX)+1
  OR B,(IX)+0       ;operation or
  LOAD (IX)+1,A
  LOAD (IX)+0,B     ;sauvetage sur la pile


CTION CALLII
------------
  LOAD SH,SP        ;charge le stack pointer dans SH-SL
  LOAD IX,SH        ;charge SH-SL dans IX
  INC SP
  INC SP            ;ajuste le stack pointer
  CALL (IX)+1       ;appel de la fonction


CTION COMPI
-----------
  POP B
  POP A             ;extrais l'operande
  CPL A
  CPL B             ;complement
  PUSH A
  PUSH B            ;sauvetage sur la pile


CTION DUPLI
-----------
  POP B
  POP A             ;extrais l'operande
  PUSH A
  PUSH B            ;sauvetage sur la pile
  PUSH A
  PUSH B            ;sauvetage sur la pile


TION EQI
--------
  POP B
  POP A             ;extrais l'operande 2
  INC IX,SP         ;IX pointe sur l'operande 1
  CALL EQI

  EQI:
  COMP B,(IX)+0     ;compare les parties high des 2 operandes
  JUMP,NE .+8       ;branche si !=
```

```
   COMP A,(IX)+1     ;compare les parties low des 2 operandes
   JUMP,EQ .+3       ;branche si ==
   CLR B
   CLR A
   LOAD (IX)+0,B
   LOAD (IX)+1,A     ;sauve le resultat sur la pile
   RET


JCTION EXORI
------------
   POP B
   POP A             ;extrais l'operande 2
   INC IX,SP         ;IX pointe sur l'operande 1
   XOR A,(IX)+1
   XOR B,(IX)+0      ;operation exor
   LOAD (IX)+0,A
   LOAD (IX)+1,B     ;sauve le resultat sur la pile


JCTION ANDI
-----------
   POP B
   POP A             ;extrais l'operande 2
   INC IX,SP         ;IX pointe sur l'operande 1
   AND A,(IX)+1
   AND B,(IX)+0      ;operation and
   LOAD (IX)+0,A
   LOAD (IX)+1,B     ;sauve le resultat sur la pile


 CTION INCAFT
-------------
   POP B
   POP A             ;extrais l'adresse au sommet de la pile
   LOAD SH,B         ;sauve B dans SH
   LOAD B,# pr       ;charge le parametre dans B
   CALL INCAFT
   PUSH A
   PUSH B            ;sauve le resultat sur la pile

   INCAFT:
   LOAD SL,A         ;sauve A dans SL
   LOAD IX,SH        ;charge l'adresse dans IX
   LOAD A,(IX)+1     ;charge la partie low de la valeur a incrementer
   PUSH A            ;sauve A sur la pile
   ADD A,B           ;addition du parametre
   LOAD (IX)+1,A     ;restitue la nouvelle valeur(partie low)
   LOAD B,(IX)+0     ;charge la partie high de la valeur dans B
   PUSH B            ;sauve B sur la pile
   ADDC B,#0         ;addition avec carry
   LOAD (IX)+0,B     ;restitue la nouvelle valeur(partie high)
   POP B
   POP A             ;extrais la valeur ancienne de la pile
   RET
```

```
CTION DECAFT
------------
  POP B
  POP A             ;extrais l'adresse au sommet de la pile
  LOAD SH,B         ;sauve B dans SH
  LOAD B,# pr       ;charge le parametre dans B
  CALL DECAFT
  PUSH A
  PUSH B            ;sauve le resultat sur la pile

  DECAFT:
  LOAD SL,A         ;sauve A dans SL
  LOAD IX,SH        ;charge l'adresse dans IX
  LOAD A,(IX)+1     ;charge la valeur a decrementer (partie low)
  PUSH A            ;sauve A sur la pile
  SUB A,B           ;calcul de la nouvelle valeur(partie low)
  LOAD (IX)+1,A     ;restitue la nouvelle valeur(partie low)
  LOAD B,(IX)+0     ;charge la valeur a decrementer dans B(partie high)
  PUSH B            ;sauve B sur la pile
  SUBC B,#0         ;calcul de la nouvelle valeur(partie high)
  LOAD (IX)+0,B     ;restitue la nouvelle valeur (partie high)
  POP B
  POP A             ;extrais l'ancienne valeur de la pile
  RET


CTION DECBEF
------------
  POP B
  POP A             ;extrais l'adresse au sommet de la pile
  LOAD SH,B         ;sauve B dans SH
  LOAD B,# pr       ;charge le parametre dans B
  CALL DECBEF
  PUSH A
  PUSH B            ;sauve le resultat sur la pile

  DECBEF:
  LOAD SL,A         ;sauve A dans SL
  LOAD IX,SH        ;charge l'adresse dans IX
  LOAD A,(IX)+1     ;charge la valeur dans A (partie low)
  SUB A,B           ;calcul de la nouvelle valeur
  LOAD (IX)+1,A     ;restitue la nouvelle valeur (partie low)
  LOAD B,(IX)+0     ;charge la valeur dans B (partie high)
  SUBC B,#0         ;calcul de la nouvelle valeur
  LOAD (IX)+0,B     ;restitue la nouvelle valeur (partie high)
  RET


TION INCBEF
-----------
  POP B
  POP A             ;extrais l'adresse au sommet de la pile
  LOAD SH,B         ;sauve B dans SH
  LOAD B,# pr       ;charge le parametre dans B
  CALL INCBEF
  PUSH A
  PUSH B

  INCBEF:
  LOAD SL,A         ;sauve A dans SL
```

```
   LOAD A,(IX)+1     ;charge la valeur dans A (partie low)
   ADD A,B           ;calcul de la nouvelle valeur
   LOAD (IX)+1,A     ;restitue la nouvelle valeur (partie low)
   LOAD B,(IX)+0     ;charge la valeur dans B (partie high)
   ADDC B,#0         ;calcul de la nouvelle valeur
   LOAD (IX)+0,B     ;restitue la nouvelle valeur
   RET


JCTION INDIRI
-------------
   POP B
   POP A             ;extrais l'adresse au sommet de la pile
   CALL INDIRI
   PUSH A
   PUSH B            ;sauve le resultat sur la pile

   INDIRI:
   LOAD SL,A
   LOAD SH,B
   LOAD IX,SH        ;charge l'adresse dans IX
   LOAD A,(IX)+1
   LOAD B,(IX)+0     ;charge la valeur dans A et B
   RET


JCTION JUMPI
-------------
   LOAD SH,SP
   LOAD IX,SH        ;charge SP dans IX
   INC SP
   INC SP  ;ajuste le stack pointer
   JUMP (IX)+1       ;branchement


CTION JUMPT
-----------
   POP A             ;extrais la partie high de l'operande
   COMP A,#0
   JUMP,NE .+7       ;branche si !=
   POP B             ;extrais la partie low de l'operande
   COMP B,#0
   JUMP,EQ .+5       ;branche si ==
   JUMP par


CTION LTI
---------
   POP B
   POP A             ;extrais l'operande 2
   INC IX,SP         ;IX pointe sur l'operande 1
   CALL LTI

   LTI:
   COMP B,(IX)+0     ;comparaison des parties high des 2 operandes
   JUMP,LT .+12      ;branche si <
   JUMP,GT .+6       ;branche si >
```

```
  COMP A,(IX)+1     ;comparaison des parties low des 2 operandes
  JUMP,LE .+6       ;branche si <=
  LOAD A,#1
  JUMP .+3
  CLR A
  CLR B
  LOAD (IX)+1,A
  LOAD (IX)+0,B     ;sauve le resultat sur la pile
  RET


CTION GEI
---------
  POP B
  POP A             ;extrais l'operande 2 dans A et B
  INC IX,SP         ;IX pointe sur l'operande 1
  CALL GEI

  GEI:
  COMP B,(IX)+0     ;comparaison des parties high des 2 operandes
  JUMP,GT .+12      ;branche si >
  JUMP,LT .+6       ;branche si <
  COMP A,(IX)+1     ;comparaison des parties low des 2 operandes
  JUMP,GT .+6       ;branche si >
  LOAD A,#1
  JUMP .+3
  CLR A
  CLR B
  LOAD (IX)+1,A
  LOAD (IX)+0       ;sauve le resultat sur la pile,B
  RET


CTION LSHFTI
------------
  POP B
  POP A             ;extrais l'operande 2 dans A - B=0
  INC IX,SP         ;IX pointe sur l'operande 1
  SLC (IX)+1
  RLC (IX)+0        ;shift d'un bit a gauche
  DEC A             ;decremente A de 1
  JUMP,ZC .-5       ;branche si !=0


CTION RSHFTI
------------
  POP B
  POP A             ;extrais l'operande 2 dans A - B=0
  INC IX,SP         ;IX pointe sur l'operande 1
  SRC (IX)+0
  RRC (IX)+1        ;shift d'un bit a droite
  DEC A             ;decremente A de 1
  JUMP,ZC .-5       ;branche si !=0


CTION LVEXTI
------------
  LOAD IX,pr        ;charge la valeur dans IX
```

```
   LOAD SH, IX
   LOAD A, SL
   LOAD B, SH
   PUSH A
   PUSH B              ; sauve le resultat sur la pile


UCTION LVEXTC
-------------
   LOAD A, pr          ; charge la valeur caractere dans A
   CLR B               ; B=0
   PUSH A
   PUSH B              ; sauve le resultat sur la pile


JCTION NEGI
-----------
   POP B
   POP A               ; extrais l'operande dans A et B
   CPL A
   CPL B               ; complement
   ADD A, #1
   ADDC B, #0          ; resultat + 1
   PUSH A
   PUSH B              ; sauve le resultat sur la pile


JCTION NOTI
-----------
   LOAD SH, SP
   LOAD IX, SH         ; charge SP dans IX
   CLR A               ; A=0
   COMP A, (IX)+2
   JUMP, NE .+6        ; branche si !=
   COMP A, (IX)+1
   JUMP, EQ .+8        ; branche si ==
   CLR (IX)+2
   CLR (IX)+1          ; mise a 0
   JUMP .+4
   INC (IX)+2          ; resultat = 1


CTION REMI
----------
   POP B
   POP A               ; extrais le diviseur
   INC IX, SP          ; IX pointe sur le dividende
   CALL DIVREM
   INC SP
   INC SP              ; ajuste le stack pointer
   PUSH A
   PUSH B              ; sauve le resultat sur la pile


CTION DIVI
```

```
----------
    POP B
    POP A               ;extrais le diviseur
    INC IX,SP           ;IX point sur le dividende
    CALL DIVI
    INC SP
    INC SP              ;ajuste le stack pointer
    PUSH A
    PUSH B              ;sauve le resultat sur la pile

    DIVI:
    CALL DIVREM         ;appel de DIVREM
    LOAD A,SSL
    COMP A,#1
    JUMP,NE .+11
    LOAD SH,IX
    LOAD A,SL
    LOAD B,SH           ;resultat dans A et B
    CALL POSIT          ;appel de POSIT
    RET


UCTION SUBTRI
-------------
    INC IX,SP           ;IX pointe sur l'operande 2
    INC SP
    INC SP              ;ajuste le stack pointer
    POP B
    POP A               ;extrais l'operande 1
    SUB A,(IX)+1
    SUBC B,(IX)+0       ;soustraction
    PUSH A
    PUSH B              ;sauve le resultat sur la pile


UCTION JNE
----------
    POP B
    POP A               ;extrais l'operande 2
    LOAD SH,A           ;sauve A dans SH
    POP A               ;extrais la partie high de l'operande 1
    COMP A,B            ;comparaison des parties high
    JUMP,NE .+7         ;branche si !=
    POP A               ;extrais la partie low de l'operande 1
    COMP A,SH           ;comparaison des parties low
    JUMP,EQ .+5         ;branche si ==
    JUMP  pr


UCTION JEQ
----------
    POP B
    POP A               ;extrais l'operande 2
    LOAD SH,A           ;sauve A dans SH
    POP A               ;extrais la partie high de l'operande 1
    COMP A,B            ;comparaison des parties high
    JUMP,NE .+10        ;branche si !=
    POP B               ;extrais la partie low de l'oprande 1
```

```
    COMP B,SH          ;comparaison des parties low
    JUMP,NE .+5        ;branche si !=
    JUMP  pr


RUCTION JLT
-----------
    POP B
    POP A              ;extrais l'operande 2
    INC IX,SP          ;IX pointe sur l'operande 1
    INC SP
    INC SP             ;ajuste le stack pointer
    COMP B,(IX)+0      ;comparaison des parties high
    JUMP,GT .+8        ;branche si >
    JUMP,LT .+9        ;branche si <
    COMP A,(IX)+1      ;comparaison des parties low
    JUMP,LE .+5        ;branche si <=
    JUMP  pr


RUCTION JGT
-----------
    POP B
    POP A              ;extrais l'operande 2
    INC IX,SP          ;ix pointe sur l'operande 1
    INC SP
    INC SP             ;ajuste le stack pointer
    COMP B,(IX)+0      ;comparaison des parties high
    JUMP,LT .+8        ;branche si <
    JUMP,GT .+9        ;branche si >
    COMP A,(IX)+1      ;comparaison des parties low
    JUMP,GE .+5        ;branche si >=
    JUMP  par


UCTION JGE
-----------
    POP B
    POP A              ;extrais l'operande 2
    INC IX,SP          ;IX pointe sur l'operande 1
    INC SP
    INC SP             ;ajuste le stack pointer
    COMP B,(IX)+0      ;comparaison des parties high
    JUMP,LT .+8        ;branche si <
    JUMP,GT .+9        ;branche si >
    COMP A,(IX)+1      ;comparaison des parties low
    JUMP,GT .+5        ;branche si >
    JUMP  pr


NE DIVREM
---------
                       ;cette routine effectue la division
                       ;de 2 nombres positifs.
                       ;le quotient se trouve dans IX
                       ;le reste est dans A et B
  DIVREM:
```

```
    CLR SSL          ;SSL=0
    CALL TESTNEG     ;appel de TESTNEG
    LOAD SL,A
    LOAD SH,B        ;sauve le diviseur dans SH et SL
    LOAD A,(IX)+1
    LOAD B,(IX)+0    ;charge le dividende dans A et B
    CALL TESTNEG     ;appel de TESTNEG
    LOAD IX,#0       ;IX=0  IX contiendra le quotient
    COMP B,SH        ;teste si le dividende est > que le diviseur
                     ; (partie high)
    JUMP,LT .+15     ;branche si <
    JUMP,GT .+6      ;branche si >
    COMP A,SL        ;teste si le dividende est > que le diviseur
                     ; (partie low)
    JUMP,LT .+9      ;branche si <
    SUB A,SL
    SUBC B,SH        ;soustrais le diviseur du dividende
    INC IX           ;IX=IX+1
    JUMP .-15
    RET


INE TESTNEG
----------
                     ;cette routine permet de tester
                     ;le signe des operandes.
                     ;si les 2 operandes sont de meme
                     ;signes,SSL vaudra 0(si tous les 2
                     ;positifs)ou 2(si tous les 2 negatifs)
                     ;elle fait aussi appel a POSIT
    TESTNEG:
    TEST B
    JUMP,GE .+7
    INC SSL
    CALL POSIT
    RET


INE POSIT
--------
                     ;cette routine permet de prendre le
                     ;complement d'un nombre negatif pour
                     ;le rendre positif.
    POSIT:
    CPL A
    CPL B
    ADD A,#1
    ADDC B,#0
    RET
```

ANNEXE 4

PROCEDURE D'UTILISATION DU CROSS-COMPILATEUR

PROCEDURE D'UTILISATION DU CROSS-COMPILATEUR

Nous expliquons ici toutes les operations a realiser pour
iliser le cross-compilateur pour le MOTOROLA 6800.
Nous expliquons pour commencer l'enchainement des differen-
s operations a realiser avant de presenter les 2 commandes permet-
nt de realiser l'entierete de ces operations.

Premiere operation:

pre-compilation VAC

Cette pre-compilation permet de traduire le programme C en texte VAC.
La commande qui realise cette pre-compilation est la commande ck et se presente comme suit:

```
ck fichier
```

Le resultat est place dans fichier.k

Deuxieme operation:

traduction en code CALM

Cette phase de traduction permet de traduire chaque instruction VAC en code CALM.
Elle est realisee par la commande:

```
sno motorola.1 <fichier.k >fichier1
```

motorola.1 est le programme de traduction
fichier1 recevra le resultat

Troisieme operation:

optimisation du code CALM

Cette phase d'optimisation realise les fonctions suivantes:

- -introduction de la sequence d'intialisation du programme.
- -introduction des procedures
- -suppresssion des sequences inutiles

Elle est realisee par la commande:

```
sno motorola.2 <fichier1 >fichier2
```

motorola.2 est le programme d'optimisation
fichier2 est le fichier resultat

Quatrieme operation:
---

introduction de la description UNIASS

Le fichier resultat de la phase d'optimisation doit etre precede de la description UNIASS du MOTOROLA 6800.

Ceci peut etre realise par:

```
ed fichier2
1r UNI6800
w
q
```

UNI6800 est le fichier qui comprend la description UNIASS pour le MOTOROLA 6800

Cinquieme operation:
---

cross-assemblage UNIASS et formatage

Cette operation permet d'assembler le programme CALM et de formater le code objet en fonction des particularites du MOTOROLA 6800

Elle est realisee par:

```
uniass.e fichier2 fichier3 >fichier4
epmot fichier3 fichier5
```

uniass.e est le programme de cross-assemblage
fichier3 contient le code objet produit par uniass.e
fichier4 contient le 'listing' d'assemblage
epmot est le programme assurant le formatage
fichier5 contient le code objet formate

Deux commandes permettent de realiser l'ensemble de ces erations:

```
sh motorola.cde1 fichier.c fichier.k
```

Cette commande fournit tous les resultats intermediaires ns les fichiers suivants:

fichier.c: programme C
fichier.k: programme VAC
fichier.k.cal: programme input de uniass.e
fichier.k.obj: programme objet non formate
fichier.k.moto: programme objet formate
fichier.k.res: 'listing' d'assemblage

```
sh motorola.cde2 fichier1.c fichier1.k fichier2
```

Cette commande fournit uniquement le 'listing' d'assemblage le programme resultat formate dans les fichiers:

fichier2.res: 'listing' d'assemblage
fichier2.moto: programme objet formate